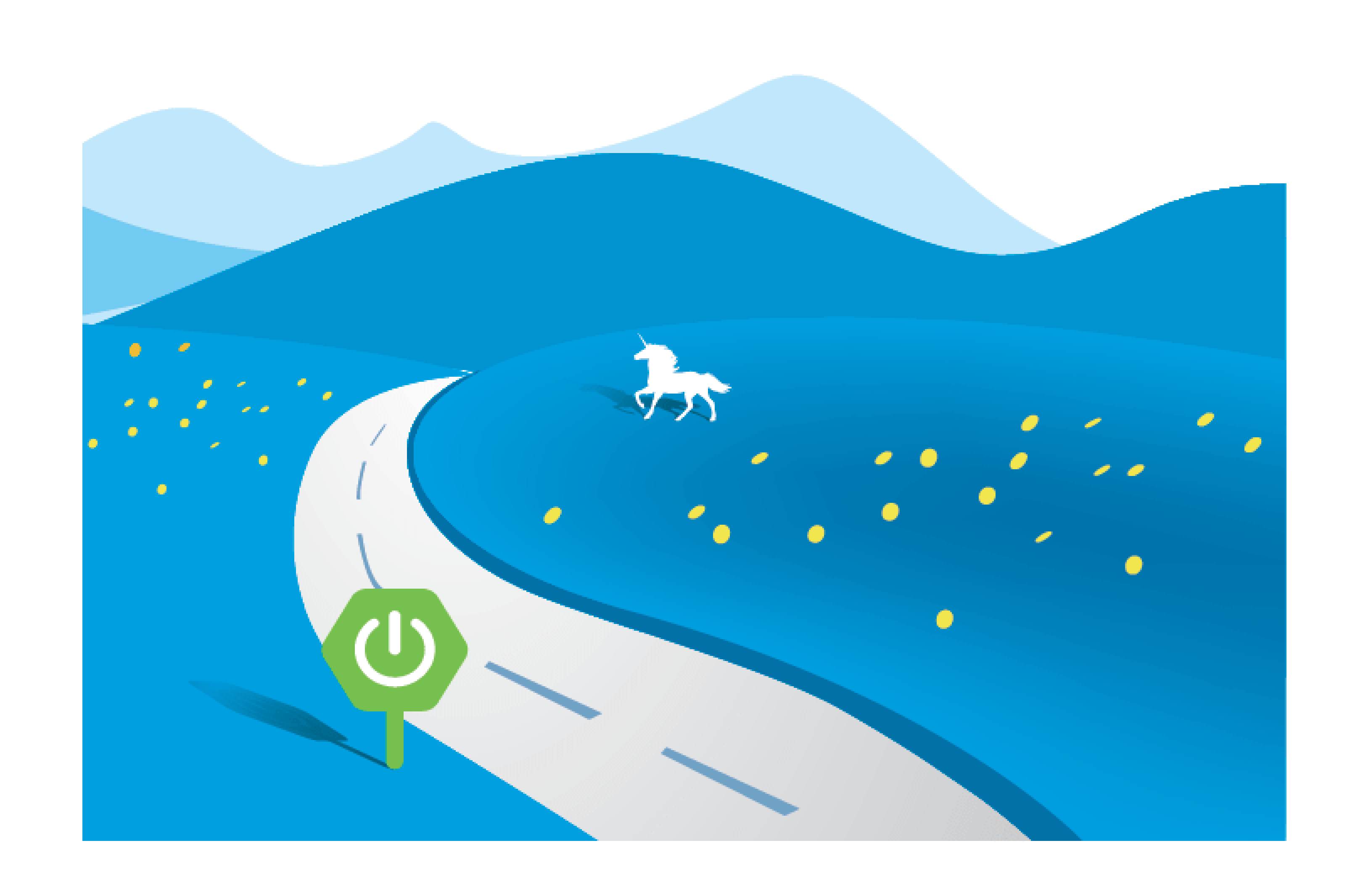




从0到1，决战Spring Boot

大老杨这本书，是我花了3天时间审校的。全书没有废话，一切从代码案例出发，记录了各种坑的解决方法，是Spring Boot初学者及核心技术巩固的最佳实践。

——泥瓦匠

Spring Boot 2 实战之旅

杨　洋　著
泥瓦匠　审校

清華大学出版社

Spring Boot 2 实战之旅

杨 洋 著



清華大学出版社
北 京

内 容 简 介

Spring Boot 框架是目前微服务框架的最佳选择之一。本书采用 Spring Boot 2.0.3 版本讲解，从零起步系统深入地剖析了 Spring Boot 的核心技术。从功能点出发，每一章都是不同的 Spring Boot 应用之旅。全书分为 14 章，第 1 章和第 2 章是学习 Spring Boot 的入门阶段，从 Spring Boot 简介到开发环境部署等，让读者对 Spring Boot 有一个初步的认识；第 3 章到第 10 章是 Spring Boot 的融合阶段，介绍了 Spring Boot 搭建 Web 项目、操作数据库、使用缓存、日志、整合安全框架、结合消息队列和搜索框架，这些都是日常开发中一定会用到的内容，经过这个阶段的学习，会让读者熟练地运用 Spring Boot 进行敏捷开发。第 11 章和第 12 章是 Spring Boot 的拓展阶段，主要介绍了 Spring Boot 的一些常用的功能和如何在实际应用中的部署。第 13 章和第 14 章是 Spring Boot 的实战阶段，经过这两章的学习，使读者对 Spring Boot 的运用更加熟练，掌握实际项目的开发技能。

本书的特点是示例代码丰富，实用性和系统性较强，读者可以直接还原书中的示例。本书适用于初学者、Java 开发人员、Spring 爱好者和架构师。

图书在版编目（CIP）数据

Spring Boot 2 实战之旅/杨洋著.—北京：清华大学出版社，2019
ISBN 978-7-302-53162-3

Ⅰ. ①S… Ⅱ. ①杨… Ⅲ. ①JAVA 语言—程序设计Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2019）第 116039 号

责任编辑：王金柱
封面设计：王　翔
责任校对：闫秀华
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**24.25　　**字　　数：**621 千字
版　　次：2019 年 8 月第 1 版　　**印　　次：**2019 年 8 月第 1 次印刷
定　　价：79.00 元

产品编号：081783-01

前　　言

微服务一词相信对很多开发者来说已经耳熟能详了。在我曾经工作的公司，还是使用单体项目来部署时，无论是打包还是运行都耗时耗力，这一直让我很苦恼。同时，每次需要创建新应用、构建项目配置 Spring 的时候也十分麻烦。一次偶然的情况，我接触了 Spring Boot 框架，开始对其“约定优先配置”的特性着迷了。这个由 Pivotal 团队进行维护开发的 Spring Boot，版本更迭非常快，社区活跃度很高。我在闲暇之余查阅了国内很多招聘网站，原来已经有很多公司将 Spring Boot 作为必备技能。

此后，我花费了很长的时间翻看技术博客、官方文档等，深入学习 Spring Boot 框架。在公司接下来的项目中，都以 Spring Boot 为主来构建项目，并且成功地将很多使用 Spring Boot 的项目投入生产，Spring Boot 框架的快速构建与部署与公司快速迭代版本的风格完美呼应。这是 Spring Boot 值得学习的一大原因。

本书沿袭我学习 Spring Boot 的路线，使用 Spring Boot 与当今常用的中间件结合，并且配备对应的实例代码。最后的两章项目实战是对 Spring Boot 的学习之路做出总结，为本书画上一个圆满的句号。希望读者阅读本书后能够有所收获。

如何阅读本书

在阅读本书的过程中，建议对照源代码按顺序学习。当然，如果对部分章节的内容比较熟悉，也可以直接跳过，学习需要巩固的章节。本书内容共分为 14 章，开发工具使用 IntelliJ IDEA，Spring Boot 版本为 2.0.3，各章节内容说明如下：

第 1 章介绍 Spring Boot 框架的特点以及学习它的重要性，最后列出 Spring Boot 的历史版本，让读者对 Spring Boot 有一个大致的了解。

第 2 章介绍如何搭建 Spring Boot 的开发环境，通过使用 IntelliJ IDEA 构建 Spring Boot 项目，并且对 Spring Boot 项目的基础结构进行介绍。

第 3 章介绍如何使用 Spring Boot 开发 Web 应用，了解 Spring MVC 和 Spring Web Flux 的不同，最后学习 Spring Boot 的一些 Web 模板框架，让读者可以对 Spring Boot 开发 Web 应用游刃有余。

第 4 章和第 5 章都是基于 Spring Boot 对数据库的使用进行学习。其中，第 4 章从 Spring Boot 使用各种数据库的依赖和配置开始介绍，然后介绍当今 Java 语言流行的 ORM 框架的使用，最后学习 Spring Boot 使用 Druid 数据库连接池。第 5 章介绍 Spring Boot 常用缓存框架，最后对 Redis 和 Memcached 进行比较，让读者选择缓存时有一定的基础。

第 6 章介绍 Spring Boot 对几种常用日志框架的使用，最后介绍分布式情况下如何使用 ELK 进行日志收集。

第 7 章介绍当今比较常用的两种安全框架，并且使用详细的案例对二者进行运用。

第 8 章介绍 Spring Boot 如何进行监控，涉及当今 Spring Boot 框架常用的监控，使读者对 Spring Boot 的运行状态更加了解。

第 9 章介绍 Spring Boot 如何使用消息队列，分别从 RabbitMQ、Kafka 和 RocketMQ 的使用实例进行介绍，最后对三者进行比较，让读者在选择消息队列时有一定的借鉴。

第 10 章对 Spring Boot 的两大常用搜索框架进行详细的介绍，从普通增、删、改、查到复杂查询，让读者使用搜索框架时不再茫然。

第 11 章介绍使用 Spring Boot 时的一些小技巧，比如启动 Banner、Lombok、邮件发送、事务、异常等。虽然知识略微零散，但是都是实用的技巧。

第 12 章介绍 Spring Boot 的多种部署方式，让读者可以根据实际情况部署自己的应用程序。

第 13 章和第 14 章分别使用博客系统和博客后台系统对 Spring Boot 的使用进行综合实战，这两个实战案例是对本书内容的总结。

本书读者对象

- *初学者*
- Java *开发人员*
- *架构师*
- Spring *爱好者*

本书技术支持

非常感谢大家能够购买和阅读本书。虽然完成本书尽了笔者最大的努力，但是由于笔者的精力和能力有限，在编写过程中难免会有一些疏漏和不足之外，希望各位读者不吝指正。关于本书的任何问题都可以发送电子邮件至 yangyang@dalaoyang.cn 与我交流。

源代码下载

本书所有源代码均上传至码云，地址是 https://gitee.com/dalaoyang/springboot_book。如果下载有问题，请发送电子邮件至 booksaga@126.com，邮件主题为“求 Spring Boot 2 实战之旅下载资源”。

致谢

在编写本书时，我得到了很多人的帮助。

首先，感谢我的妻子，在我遇到困难时给予鼓励，在我迷茫时的开导，谢谢她在我编写本书的过程中承担了所有家务，并且不遗余力地支持我。

其次，感谢我的父母，感谢他们从小对我的抚育与培养，感谢他们对我事业的支持。

另外，还需要感谢一下泥瓦匠在百忙之中对本书的细心校对，让本书的一些细节更加完善。

最后，感谢清华大学出版社的王金柱编辑，感谢您在本书编写、出版整个过程中的辛勤付出。也要感谢清华大学出版社所有参与本书编辑和出版的老师们，感谢大家对本书的帮助。

杨　洋
2019 年 3 月 1 日

目　　录

第 1 章

Spring Boot 概述

本章将对 Spring Boot 进行整体的介绍，从 Spring Boot 的特点开始，逐步让大家了解学习 Spring Boot 会为我们带来的好处，最后从 Spring Boot 的发展史（Spring Boot 1.x 到 Spring Boot 2.x）来全面概述 Spring Boot 微服务框架的多功能性。

1.1 Spring Boot 简介

Spring Boot 是由 Pivotal 团队在 2014 年发布的全新框架。从 Spring Boot 的 Logo 中可以看到，Spring Boot 是要打造一个快速构建的 Spring 应用，如图 1-1 所示。正如 Spring 官网（官网地址：http://spring.io/）介绍的：

```
    “Spring Boot is designed to get you up and running as quickly as possible,
with minimal upfront configuration of Spring. Spring Boot takes an opinionated
view of building production ready applications.”
```



图 1-1　Spring Boot 官方 Logo 图片

翻译过来的意思大概是：Spring Boot 的设计是可以尽可能快地启动和运行，只需要最少的 Spring 配置。Spring Boot 对构建生产就绪应用程序具有独特的方式。从官方的介绍可以看出 Spring Boot 的核心思想是“约定优先于配置（Convention Over Configuration）”，其本质其实还是基于

Spring 来实现的。对于了解 Spring 的人或者使用过 Spring 的人来说，Spring 烦琐的配置让很多程序员眼花缭乱（各种 XML、Annotation 配置等），甚至很多时候发生错误也很难快速定位错误的地方。而在 Spring Boot 框架中，为我们提供了默认的配置，从而使开发人员不再需要定义样板化的配置，通过这种方式，Spring Boot 致力于在蓬勃发展的快速应用开发领域（Rapid Application Development）成为领导者。

1.2 Spring Boot 的特点

Spring 团队曾经为开发者提供了无数的便利，其提供的 IOC 和 AOP 两大特性一直为广大开发者所“深爱”。当然，Spring 框架还提供了很多优秀的特性，在这里就不一一介绍了。但是，在传统 Spring 框架中有一个重大的缺点，那就是在配置的时候很复杂，需要重复地进行一些配置。Spring 团队可能感受到了这一点，在 2014 年，Spring 团队发布了 Spring Boot 框架。另外，官网首页的 Spring Boot 部分也介绍了诸多 Spring Boot 的特点，如图 1-2 所示。本节将逐一介绍 Spring Boot 框架的特点。

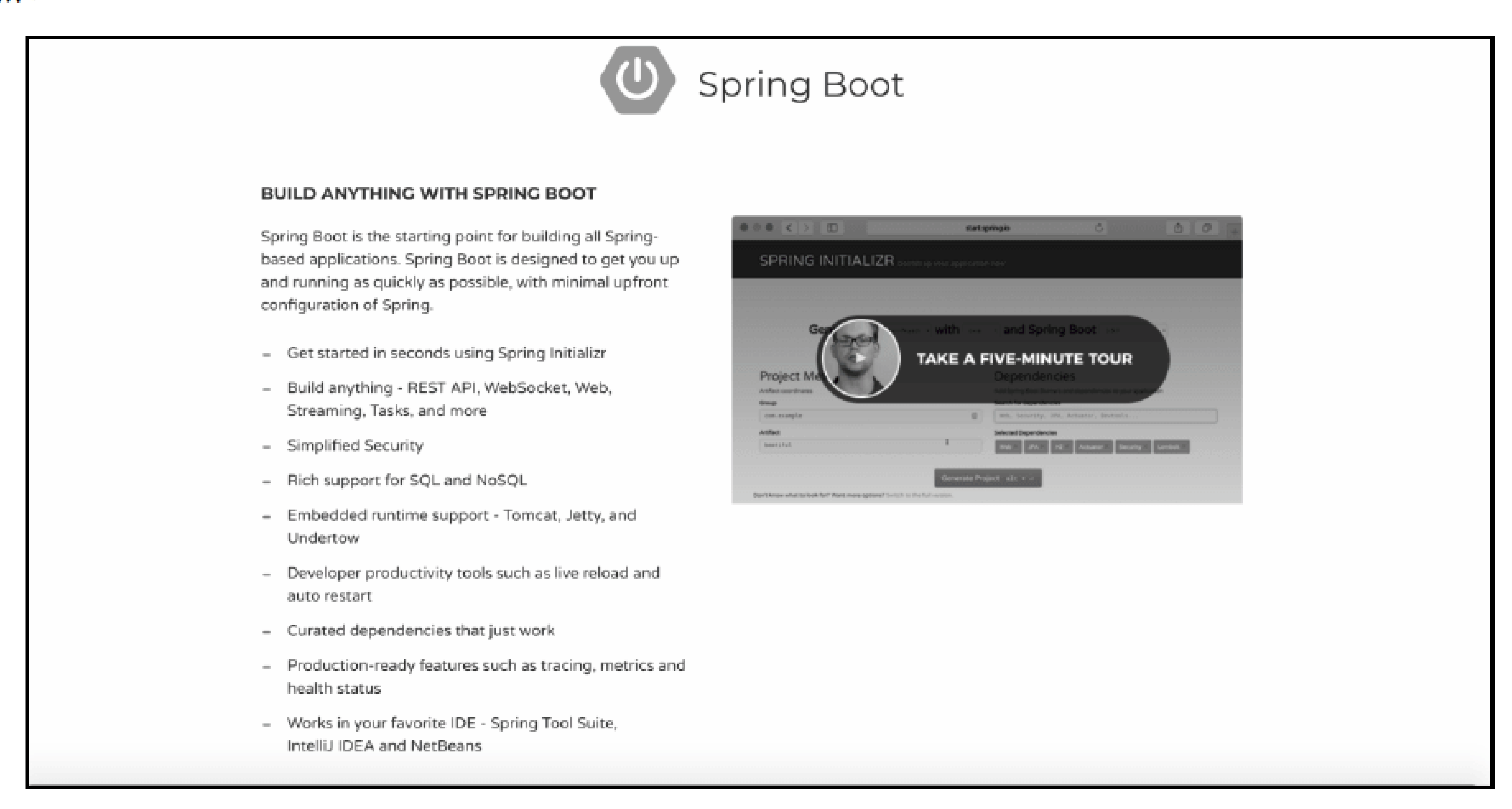


图 1-2 Spring Boot 官网简介（图片来源于 Spring 官网：http://spring.io/）

1.2.1 快速构建项目

Spring Boot 具有多种快速构建项目的方式，如下面几种形式：

（1）使用 Eclipse（MyEclipse）可以利用创建 Maven 项目的方式创建 Spring Boot 项目。当然，如果在 Eclipse 中安装了 Spring Tools，就可以直接创建 Spring Starter Project。

（2）使用 IntelliJ IDEA，可以利用创建 Spring Initializr 的方式创建 Spring Boot 项目，在后续章节会详细介绍这种方式的过程。

（3）使用 Spring Tool Suite，可以直接新建 Spring Starter Project 项目，过程类似 Eclipse 创建 Spring Boot 项目。

（4）使用官方文档创建项目，在 Spring 官方文档上面提供了一种在线生成 Spring Boot 项目的方式，首先访问 Spring 官方快速构建地址（官网地址：https://start.spring.io/），在这个页面上选择对应版本、构建工具等，填写完成后单击 Generate Project 按钮，即可在本地下载一个 Spring Boot 项目的压缩包。

当然，可能还有很多方式快速构建项目，这里就不一一介绍了。笔者在这里推荐使用 IntelliJ IDEA 开发项目，个人感觉这个开发工具还是很强大的，并且提供了很多插件供开发者使用，读者也可以根据自己的喜好进行选择，毕竟适合自己的才是最好的。

1.2.2　嵌入式 Web 容器

在传统 Java Web 项目中，当项目开发完成之后，还需要配置所需的 Web 容器（诸如 Tomcat 或者 WebLogic 之类的 Web 容器）。但是在 Spring Boot 搭建的项目中，内部提供了几种 Web 容器，如 Tomcat、Jetty 和 Undertow。在 Spring Boot 1.x 中默认为 Tomcat；Spring Boot 2.x 中则分为两种情况，引入 spring-boot-starter-web 依赖为 Tomcat，引入 spring-boot-starter-webflux 依赖则为 Netty。当然，也支持使用之前指出的几种 Web 容器，开发者只需要根据场景选择适合的 Starter 来获取一个默认配置好的容器即可，当启动成功后，应用一个默认端口为 8080 的 HTTP 服务。

1.2.3　易于构建任何应用

Spring Boot 提供了一个强大的 starter 依赖机制，实质上 Spring 团队将 Spring Boot 框架整合了一切常用的 maven 依赖，使 Spring Boot 想要整合对应依赖，就要将需要的依赖全部引入。比如，需要在项目中使用 Web，也就是我们常说的 Spring MVC，如果是原有的 maven 项目，就需要引入很多依赖才能完成这个简单的需求。但是在 Spring Boot 项目中，我们只需要在 maven 依赖中加入 spring-boot-starter-web 依赖即可，是不是很简单？这里再举一个例子，比如项目中需要使用 MySQL 数据库，这里只需要加入 MySQL 依赖，并且在配置文件中配置数据库信息就可以正常使用。

1.2.4　自动化配置

这个特点是上一个特点的延伸，在应用程序中引入依赖之后，其实还有一个强大之处在于 Spring Boot 应用会根据引入的依赖提供一些默认的配置供我们使用，如果需要修改，那么只需要在配置文件中修改对应的配置即可完成需求。这里还是以 Spring MVC 为例，传统 Spring MVC 项目需要配置对应的诸如 ApplicationContext.xml（Spring 配置文件）、ApplicationContext-mvc.xml（Spring MVC 配置文件），而在 Spring Boot 中，这些需要的配置已经为我们默认配置了一套，不需要再进行配置了。比如，我们要加入 Web 应用程序根路径 test 的话，只需要在 application.properties（Spring Boot 应用程序默认配置文件）中加入 server.servlet.context-path=/test 即可。

1.2.5 开发者工具

在开发 Web 应用的时候，总会有一个困扰我们的问题，修改代码总是伴随不断重启项目，需要不断地断开 Web 容器，再重启来测试我们的代码。在 Spring Boot 应用中提供了开发者工具（spring-boot-devtools），当我们重新编译类文件的时候，开发者工具会自动替我们重启应用，无须手动单击重启。

1.2.6 强大的应用监控

在生产环境中，应用的各项指标监控更是必不可少。在 Spring Boot 应用中提供了一个 spring-boot-starter-actuator（以下简称 Spring Boot-Actuator）来供我们查看应用的各项指标，如 health（健康检查）、dump（活动线程）、env（环境属性）、metrics（内存，CPU 等）等指标，以监控我们的应用，同时可以配合使用 spring-boot-admin-starter-server（以下简称 Spring Boot-Admin）监控我们的项目。Spring Boot-Admin 可以在利用监控 Spring Boot-Actuator 端点的同时监控所有微服务应用的健康状态，如果出现异常，就可以向维护人员发送邮件或者以其他方式给予告警。不只是这样，就连监控神器 Prometheus 也可以通过简单的配置接入 Spring Boot 应用程序中。

1.2.7 默认提供测试框架

Spring Boot 应用在创建项目之后会默认为我们创建测试类的文件，实质上就是引入 spring-boot-starter-test 依赖，然后可以通过它对各种场景进行测试，足够满足对项目的测试需求。

1.2.8 可执行 Jar 部署

由于 Spring Boot 项目内嵌 Web 容器，因此提供了一种特殊部署方式，可以直接利用 Maven 或者 Gradle 对 Spring Boot 项目进行打包，生成一个 JAR 文件，然后直接在具备环境的服务器或本地环境中利用 java -jar xx.jar 执行 JAR 文件，使应用能够快速运行。

1.2.9 IDE 多样性

正如 1.2.1 小节介绍的，Spring Boot 支持的开发工具很多，无论是曾经几乎所有开发者都使用的 Eclipse 一族，还是现在流行的 IntelliJ IDEA，又或者是专门为开发 Spring 系列而生的 Spring Tool Suite 都是开发 Spring Boot 应用的不二法宝。

1.3　为什么要学习 Spring Boot

为什么要学习 Spring Boot 呢？这可能是很多读者心中的疑惑。在 1.2 节，我们通过了解 Spring Boot 的特点应该已经对 Spring Boot 框架产生了一定的兴趣，接下来笔者将从几个方面来整体阐述学习 Spring Boot 框架的理由。

1.3.1　简化工作

Spring Boot 最大的优点是在一定程度上简化了我们的工作，大致分为以下几个角度对工作进行简化。

- 依赖简化：Spring Boot 自有的 starter 中提供了一些可以快捷使用的依赖，让整合或集成一些常用的功能更便捷。
- 配置简化：在配置方法中，如果没有特殊情况，Spring Boot 为我们提供了一些默认的配置，比如端口号默认为 8080 等。
- 部署简化：正如前面介绍的可执行 JAR 部署，与传统服务的 Web 模式部署（打 WAR 包部署）相比，连安装 Web 容器的时间都节省了，不只是开发者，对运维人员也是福音。
- 监控简化：可以通过引用 Spring Boot 依赖的方式快捷提供监控端点，无代码侵入，十分便捷。

1.3.2　微服务时代

"微服务"一词最早是由 Martin Fowler 的《Microservices》一文（原文链接为 https://martinfowler.com/articles/microservices.html）提出的，其核心思想是将一个单体应用根据业务功能拆分成为多个服务，使业务代码之间不再耦合。接下来，我们介绍一下由单体应用转变为微服务应用的好处。

1. 微服务的优势

- 服务解耦：将单体应用转变为多个服务，服务与服务之间通过 HTTP 协议或其他约定好的协议等进行网络通信。
- 技术选型广泛：微服务不需要局限于固定的技术栈，各个服务的开发团队可以根据场景自由选择开发技术，如 Java、PHP、Node.Js 等。
- 服务并行开发：可以多个团队分别开发不同的模块，每个团队负责一个或者几个服务，相互不受影响。
- 单一职责：不同服务的团队只需要关注自己团队的业务，无须经常为了熟悉业务而耽误时间。
- 独立部署：由于每个服务都是独立的项目，因此当服务开发完成后，可以直接独立部署。
- 敏捷开发：每个服务的业务迭代只需要更新对应服务的功能即可，支持快速迭代。

- 故障隔离：在传统单体项目中，如果某个功能发生故障，就可能导致整个服务发生宕机，但是在微服务中，一个服务发生宕机，其他服务仍然可以继续工作。

2. 微服务的劣势

- 部署需要花费更多的精力：当服务拆分得非常多的时候，可能需要消耗更多的精力去运维管理这些应用。传统的单体应用下，运维人员只需要保证一个服务正常运行即可，但是拆分微服务后，可能需要保持几十，甚至上百、上千的服务高效运行，这对运维人员来说是一个很大的挑战。
- 服务间的接口问题：正因为拆分了微服务，服务与服务间使用接口进行相互调用，当一个接口需要改变格式或者增减数据时，所有调用这个接口的服务都需要做出相应的调整才能正确地使用。
- 高可用：拆分微服务后，可能有很多服务需要调用同一个服务或者接口，这个服务的可用性就需要让我们更加注意。
- 分布式事务：微服务系统各个服务间可能使用不同的数据库，比如搜索服务使用 Elasticsearch、基础服务使用 MySQL、评论服务使用 MongoDB，对于不同数据库间数据的一致性将是我们面临的重大挑战。
- 网络复杂性：由于各个服务间使用接口调用，因此系统间需要考虑很多网络延迟等客观因素来保证服务间的正常运转。
- 测试的复杂性：在测试方面，服务间的接口调用、服务间的测试需要一套整体的测试方案，自动化测试就显得必不可少。

由于 Spring Boot 项目可以提供快速开发、测试、部署，因此 Spring Boot 是微服务应用的不二选择。

1.3.3 社区背景强大

社区背景强大其实是 Spring Boot 如今盛行的原因。众所周知，Spring 家族对于开发者提供了无尽的便利，而作为 Spring 的亲儿子“Spring Boot”则继承了一切 Spring 的优点，并且规避了很多 Spring 框架臃肿的缺点，而后续 Spring 家族的分布式框架 Spring Cloud 也是基于 Spring Boot 框架实现的框架，所以作为 Spring 的爱好者，或者将要学习 Spring Cloud 框架的开发者，Spring Boot 是必须要学习的。

1.3.4 市场需求

在写这本书之前，笔者游历于各大国内、国外技术论坛，无论是在国内还是在国外，Spring Boot 的呼声都特别高，而且框架的更新频率特别快。如图 1-3 所示是从 2014 年到 2018 年 Spring Boot 的百度搜索指数。

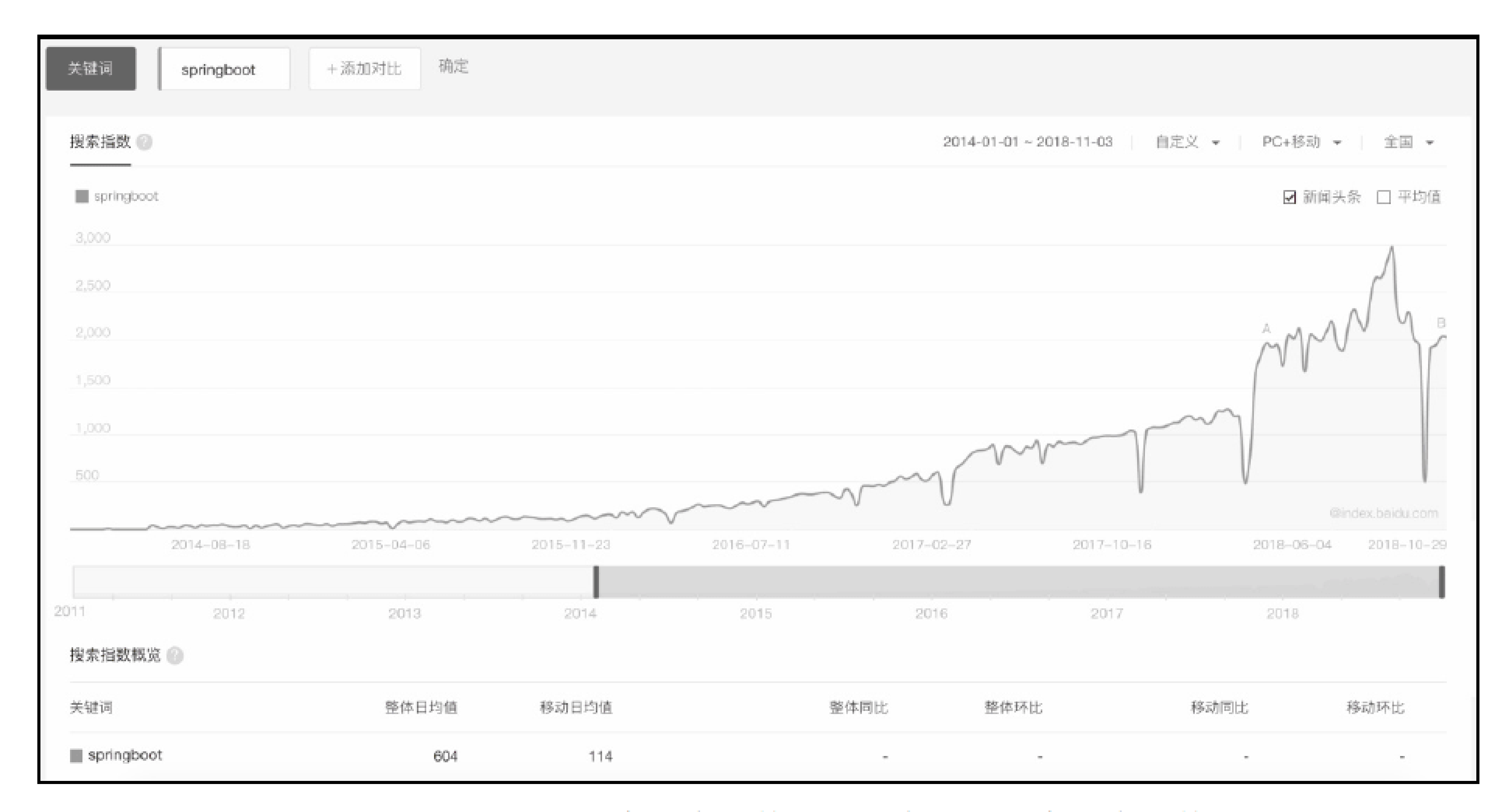


图 1-3　Spring Boot 百度搜索指数（图片来源于百度搜索指数）

从搜索指数可以看出，Spring Boot 的搜索值日趋增长，关注度特别高。

另外，我们从互联网招聘网站上来看，已经有超过 7 成以上的公司将 Spring Boot 框架作为筛选人员的必要条件，所以无论是从个人提升，还是比较实际的跳槽、涨薪等，学习 Spring Boot 都会为你的技术栈增光添彩。

1.4　Spring Boot 的发展历史

Pivotal 团队对于 Spring Boot 更新得非常频繁，而且在 Github 和国内社区的关注度都极高。接下来我们看一下 Spring Boot 的发展史。

1.4.1　发布里程碑（2013.8.6）

Phil Webb 在 Spring 官网博客上宣布了一个名为 Spring Boot 的新项目的第一个里程碑版本。

1.4.2　Spring Boot 1.0（2014.4）

Spring Boot 问世，为所有 Spring 开发提供快速和可广泛访问的入门体验，其中版本功能包括但不限于以下几点：

- 嵌入式服务器。
- 外部配置。
- 健康检查。
- 安全性。
- 快速运行。

1.4.3 Spring Boot 1.1（2014.6）

第一次更新，下面列出比较重要的几点更新，详细版本内容可以查看 Spring Boot 的 Github 官方版本介绍，地址为 https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-1.1-Release-Notes。

- 对 spring-boot-starter-test 进行修改。
- 新增对 Elasticseach 和 apache solr 的自动配置支持。
- 新增框架模板 Freemarker、Groovy 和 Velocity。
- Spring-WS 适用于 Spring Web 服务支持。
- 对 Jackson JSON 库进行了改进。
- 添加了新的注解。

1.4.4 Spring Boot 1.2（2015.3）

对之前的版本进行了修订，包括但不限于以下更新，详细版本内容可以查看 Spring Boot 的 Github 官方版本介绍，地址为 https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-1.2-Release-Notes。

- 使用 Tomcat 8 和 Jetty 9 作为嵌入式 Servlet 容器，提供 Servlet 3.1 和增强的 WebSocket 支持。
- Spring 4.1。
- 支持 JTA 实务。
- 提供 JMS 支持。
- 提供电子邮件支持。

1.4.5 Spring Boot 1.3（2016.12）

对之前的版本进行了修订，包括但不限于以下更新，详细版本内容可以查看 Spring Boot 的 Github 官方版本介绍，地址为 https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-1.3-Release-Notes。

- Spring 更新至 4.2。
- Spring Security 更新至 4.0。
- 新增 spring-boot-devtools（热部署）。
- 新增 OAuth 2 的支持。
- 缓存自动配置。

1.4.6 Spring Boot 1.4（2017.1）

对之前的版本进行了修订，包括但不限于以下更新，详细版本内容可以查看 Spring Boot 的

Github 官方版本介绍，地址为 https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-1.4-Release-Notes。

- Spring 更新至 4.3。
- Hibernate 更新至 5.0。
- 提供新的测试模块。
- Neo4J 和 Narayana 事务管理器，Caffeine cache、Elasticsearch Jest 支持。

1.4.7 Spring Boot 1.5（2017.2）

对之前版本进行了修订，包括但不限于以下更新，详细版本内容可以查看 Spring Boot 的 Github 官方版本介绍，地址为 https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-1.5-Release-Notes。

- 修改了一些 starter 的命名。
- OAuth 2 资源过滤器。
- 新的记录器端点。
- 提供 Apache Kafka、LDAP 支持。

1.4.8 Spring Boot 2.0（2018.3）

Spring Boot 2.x 版本对 Spring Boot 进行了重大的改进，官网介绍如图 1-4 所示。该版本对之前的版本进行了修订，包括但不限于以下更新，详细版本内容可以查看 Spring Boot 的 Github 官方版本介绍，地址为 https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.0-Release-Notes。

- 基于 Java 8，支持 Java 9。
- 支持 Quartz 调度程序。
- 大大简化了安全自动配置。
- 支持嵌入式 Netty。
- Tomcat、Undertow 和 Jetty 均已支持 HTTP/2。
- 全新的执行器架构，支持 Spring MVC、WebFlux 和 Jersey。
- 使用 Spring WebFlux/WebFlux.fn 提供响应式 Web 编程支持。
- 为各种组件的响应式编程提供了自动化配置，如 Reactive Spring Data、Reactive Spring Security 等。
- 用于响应式 Spring Data Cassandra、MongoDB、Couchbase 和 Redis 的自动化配置和启动器 POM。
- 引入对 Kotlin1.2.x 的支持，并提供了一个 runApplication 函数，让你通过惯用的 Kotlin 来运行 Spring Boot 应用程序。更多信息请参阅参考文档中对 Kotlin 的支持部分。
- 启动时的 ASCII 图像 Spring Boot Banner 现已支持 GIF。

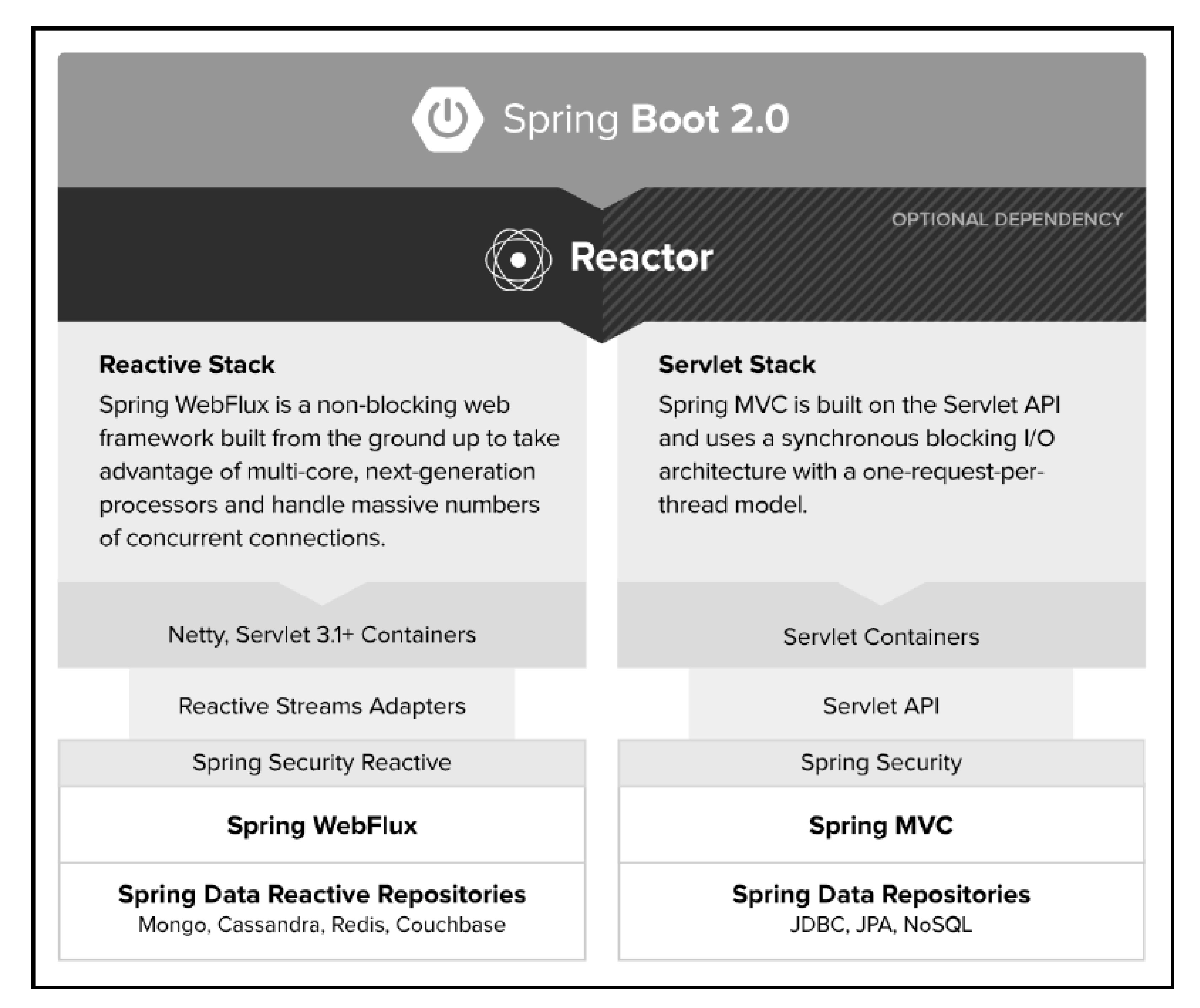


图 1-4　Spring Boot 2.0 的改动（图片来源于 Spring 官网：http://spring.io/）

1.5　小　　结

本章从多个角度带领大家了解 Spring Boot，并且介绍了 Spring Boot 历史版本的更迭以及学习它会给我们带来的好处，内容可能略显乏味。第 2 章将带领大家构建开发 Spring Boot 的环境，让我们走进 Spring Boot。

第 2 章

走进 Spring Boot

第 1 章介绍了 Spring Boot 的特点及发展历史，阅读起来可能会有一些枯燥，但是间接地映射出了尽快学习掌握 Spring Boot 的重要性。本章将带领大家搭建 Spring Boot 的开发环境，并且引领大家简单地构建项目，以及对项目的简单使用。

2.1 环 境 搭 建

正所谓“工欲善其事，必先利其器”，正如我们学习 Java 时一样，先要搭建环境，才能真正进行开发和部署。所以，本节将对 Spring Boot 的开发环境进行搭建，第一个需要安装的是 JDK。当今主流的 Java 开发工具有 Eclipse、IntelliJ IDEA、Spring Tool Suite 以及 MyEclipse 等。本书中的实例全部使用 IntelliJ IDEA 作为 IDE 进行开发，使用 Apache Maven 构建项目。

2.1.1 JDK 安装

本书中使用的是 Spring Boot 2.0.3 版本，Spring Boot 2.x 以上版本需要 JDK 1.8 以上，所以我们需要去官网下载 JDK1.8 以上的版本。登录官网（http://www.oracle.com/technetwork/java/javase/downloads/index.html），下载一个适合自己系统的 JDK1.8 安装包，然后进行安装。安装完成之后，配置一下环境变量，然后在终端或者 Windows 控制台（CMD）输入查看 JDK 版本的命令，如代码清单 2-1 所示。

代码清单 2-1 查看 JDK 版本

```
java -version
```

如果出现如图 2-1 所示的界面，就证明 JDK 安装成功了。

```
dalaoyang — -bash — 80×24
dalaoyang-3:~ dalaoyang$ java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
dalaoyang-3:~ dalaoyang$
```

图 2-1 JDK 安装成功查看图

2.1.2 IntelliJ IDEA 的安装

登录 IntelliJ IDEA 官网（下载地址：https://www.jetbrains.com/idea/download/），如图 2-2 所示，下载合适的版本，笔者使用的是 2018.1 版本，具体可以根据自己的情况下载，各版本之间的差异不是很大，下载完成之后安装即可（需要注意的是，IntelliJ IDEA 是一款收费软件，大家可以申请免费使用一年，根据自己的喜好选择 IDE）。

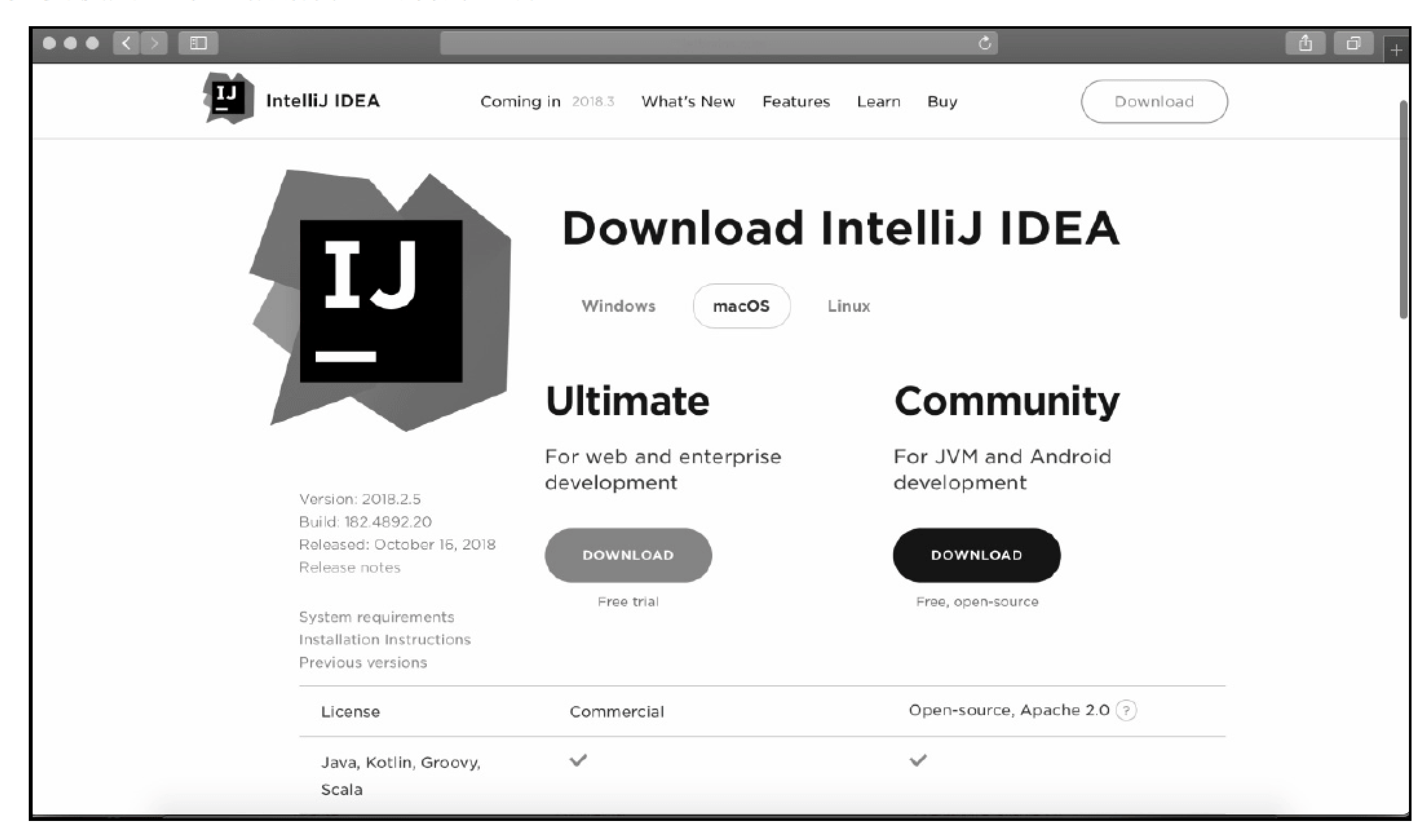


图 2-2 IntelliJ IDEA 官网下载页面

2.1.3 Maven 的安装

Maven 是一个比较常用的项目管理工具，同时提供了出色的应用程序构建能力。通常可以通过几行命令构建一个简单的应用程序。

由于本书是使用 Maven 构建应用的，因此我们需要安装一个 Maven 管理工具。通过登录 Apache Maven 官网（Apache Maven 官网地址：https://maven.apache.org/download.cgi）下载一个 3.2 以上的版本，如图 2-3 所示。

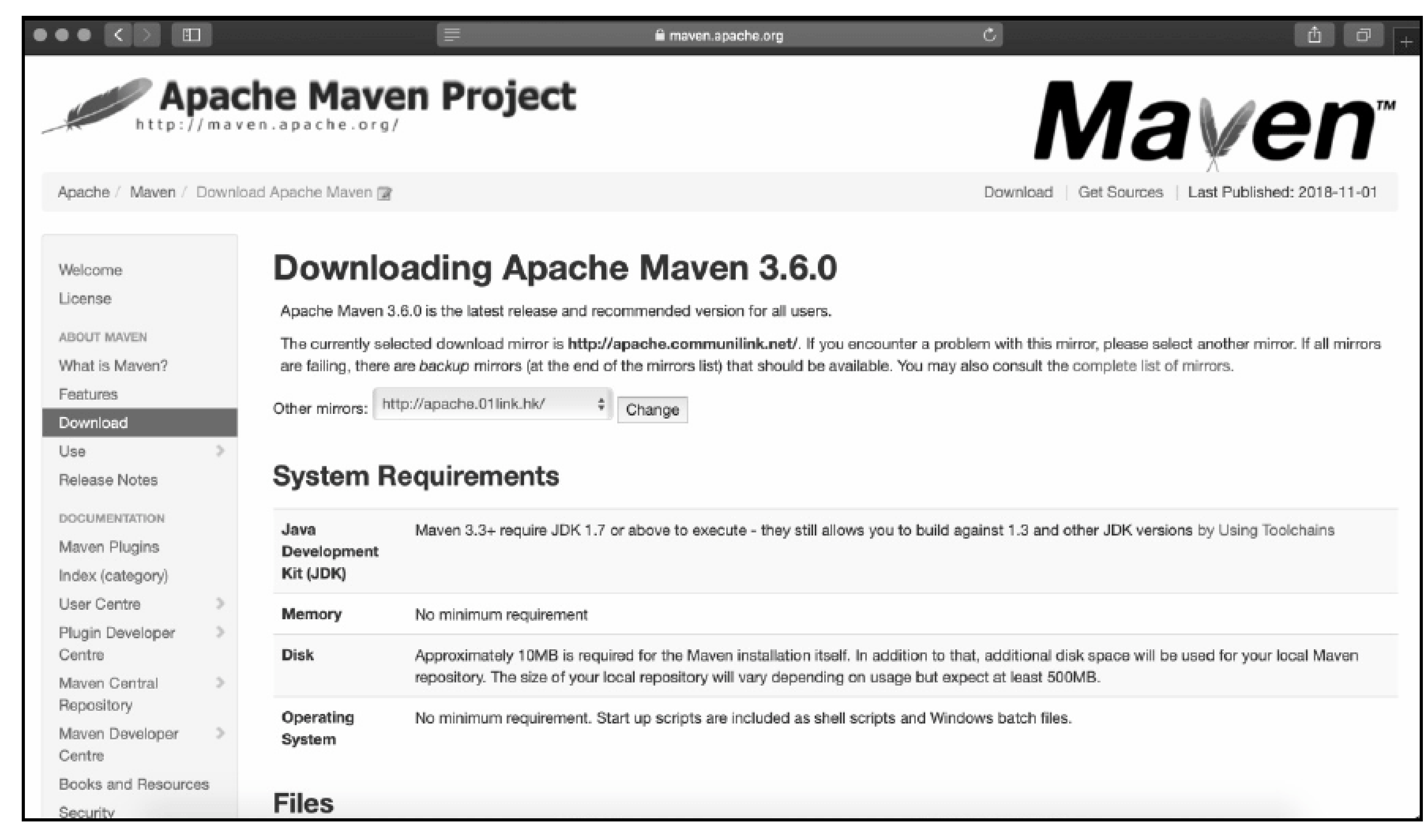


图 2-3　在 Apache Maven 官网下载

可以根据系统下载压缩包，下载之后在本地解压，解压完成后需要配置一下 Maven 环境变量，配置完成后在终端或者 Windows 控制台（CMD）输入查看 Maven 版本的命令，如代码清单 2-2 所示。


代码清单 2-2　查看 Maven 版本

```
mvn -v
```

如果出现如图 2-4 所示的界面，就证明 Maven 环境变量配置成功了。

```
apache-maven-3.5.2 — -bash — 80×24
dalaoyang-3:apache-maven-3.5.2 dalaoyang$ mvn -v
Apache Maven 3.5.2 (138edd61fd100ec658bfa2d307c43b76940a5d7d; 2017-10-18T15:58:13+08:00)
Maven home: /Users/dalaoyang/apache-maven-3.5.2
Java version: 1.8.0_131, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/jre
Default locale: zh_CN, platform encoding: UTF-8
OS name: "mac os x", version: "10.14", arch: "x86_64", family: "mac"
dalaoyang-3:apache-maven-3.5.2 dalaoyang$
```

图 2-4　查看 Maven 环境变量

Maven 环境变量到这里已经配置完成了。但是在默认情况下，Maven 下载 JAR 可能会有一些慢，可以修改为国内阿里云等下载地址，如代码清单 2-3 所示，这是笔者 Maven 的配置（settings.xml 配置），可以根据需求自行修改。

代码清单 2-3　Maven settings.xml 配置

```
<?xml version="1.0"?>
<settings>
<localRepository>/Users/dalaoyang/maven_repository</localRepository><!--
需要改成自己的 Maven 的本地仓库地址-->
```

```
    <mirrors>
        <mirror>
            <id>alimaven</id>
            <name>aliyun maven</name>
            <url>http://maven.aliyun.com/nexus/content/groups/public/ </url>
            <mirrorOf>central</mirrorOf>
        </mirror>
    </mirrors>
  <profiles>
    <profile>
       <id>nexus</id>
        <repositories>
            <repository>
                <id>nexus</id>
                <name>local private nexus</name>
                <url>http://maven.oschina.net/content/groups/public/</url>
                <releases>
                    <enabled>true</enabled>
                </releases>
                <snapshots>
                    <enabled>false</enabled>
                </snapshots>
            </repository>
        </repositories>

        <pluginRepositories>
            <pluginRepository>
            <id>nexus</id>
            <name>local private nexus</name>
            <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
            </pluginRepository>
        </pluginRepositories>
    </profile></profiles>
</settings>
```

2.1.4　IntelliJ IDEA 内配置 JDK 和 Maven

1. 配置 JDK

打开 IntelliJ IDEA，在菜单栏单击 Project Structure，然后在左侧单击 SDKs，单击+号，选择自己刚刚安装的 JDK 即可，如图 2-5 所示。

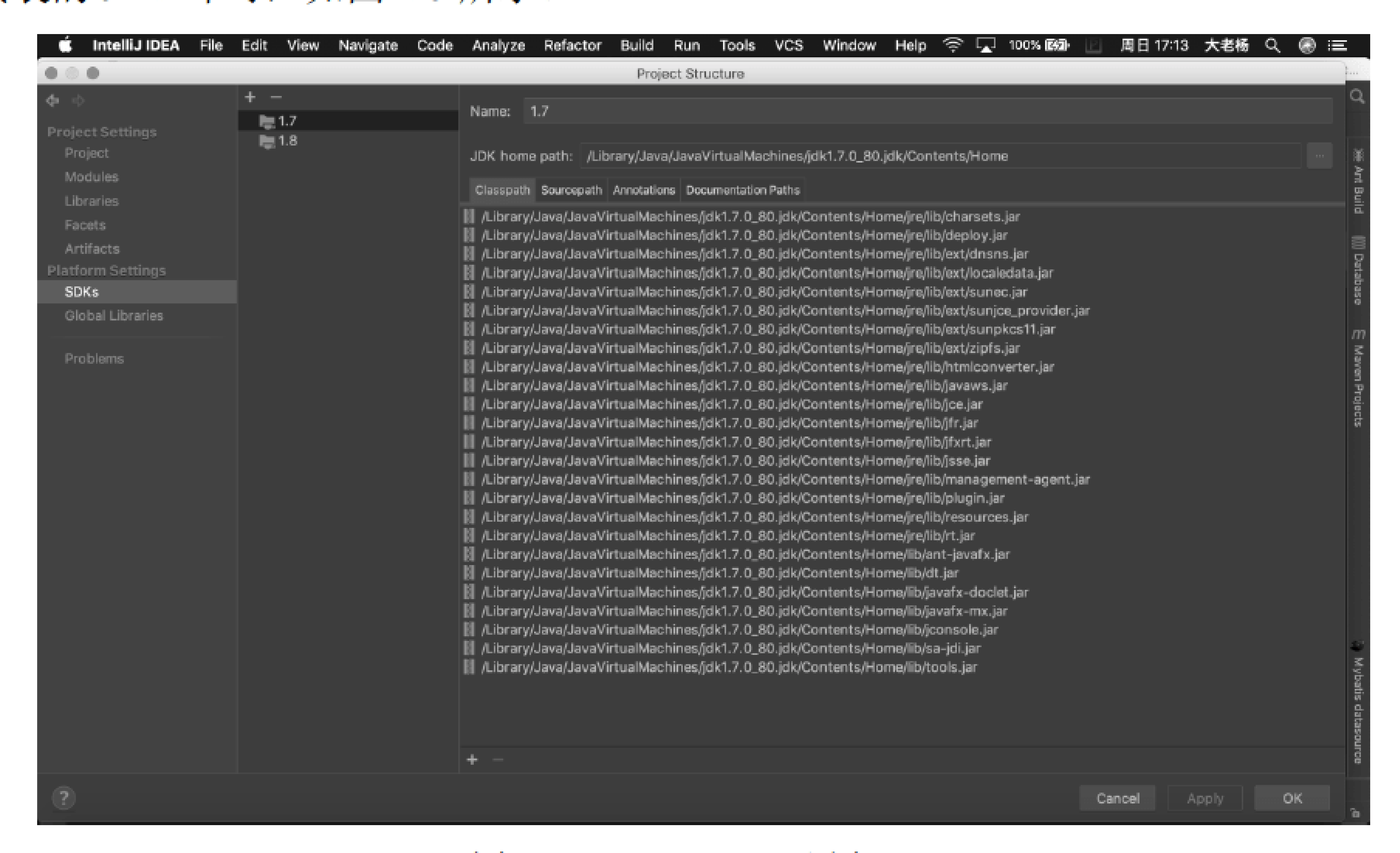


图 2-5　IntelliJ IDEA 配置 JDK

2. 配置 Maven

在菜单栏单击 Preferences，在搜索栏搜索 maven，单击查询到的 Maven，在右侧配置刚刚安装的 Maven，如图 2-6 所示。

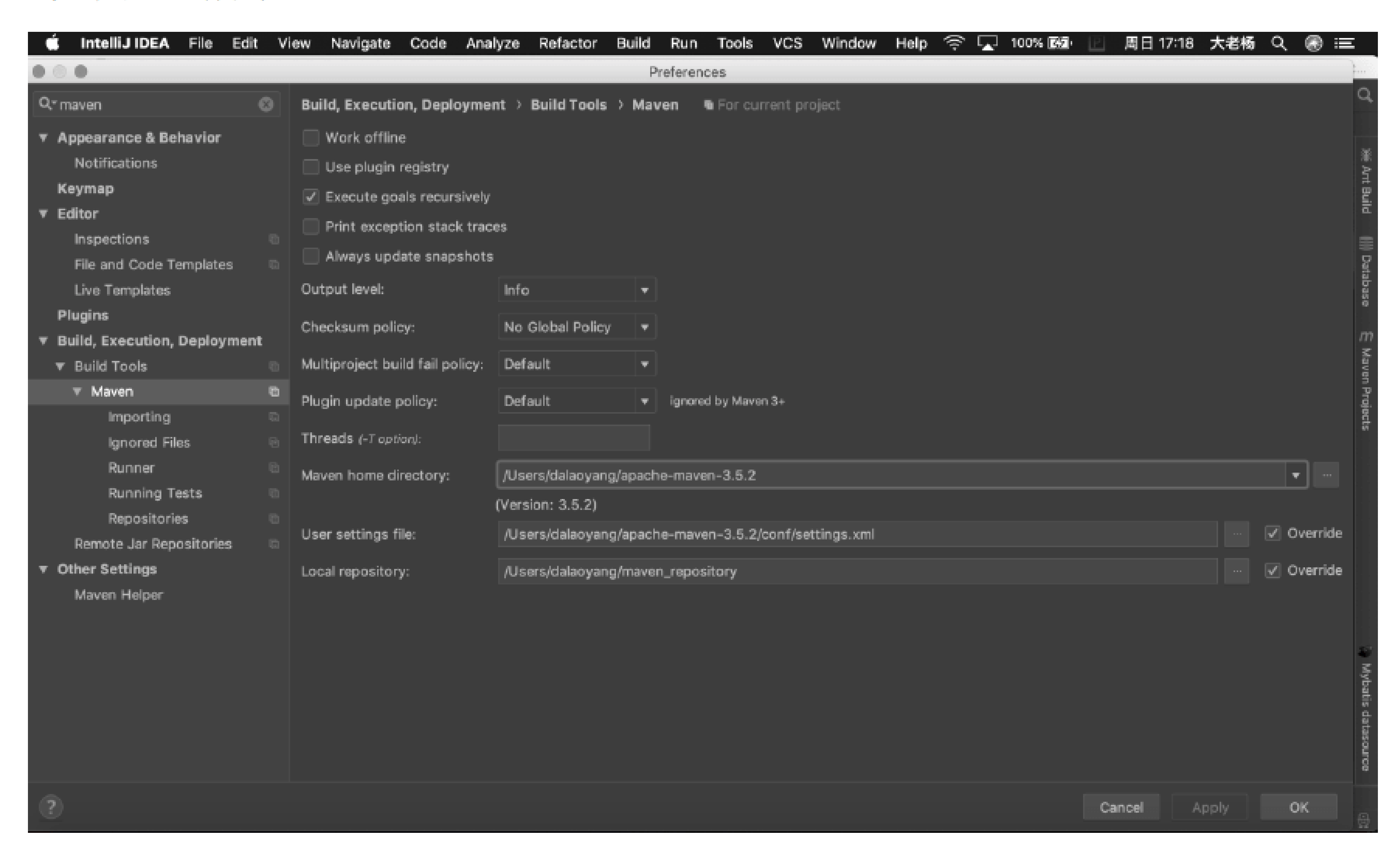


图 2-6　IntelliJ IDEA 配置 Maven

其中，需要配置以下三项。

- Maven home directory：选择刚刚安装的 Maven。
- User settings file：配置安装的 Maven 中 conf 目录下的 settings.xml。
- Local repositroy：配置 Maven 下载 JAR 包的位置。

到这里，所有配置和搭建都已经完成了。

2.2 新建 Spring Boot 项目

本节将介绍 IntelliJ IDEA 使用 Spring Initializr 创建 Spring Boot 项目。

2.2.1 开始创建项目

第一次打开 IntelliJ IDEA 或者之前关闭了 Intellij IDEA，可以看到如图 2-7 所示的界面，其中可以创建项目、导入项目、打开项目或者从版本管理工具内导入项目。

如果你之前打开过项目，重新打开 Intellij IDEA 会默认打开上次打开的项目。当然，我们也可以在 IDEA 的菜单栏选择 New 进行创建应用的操作，如图 2-8 所示。



图 2-7 IntelliJ IDEA 欢迎界面

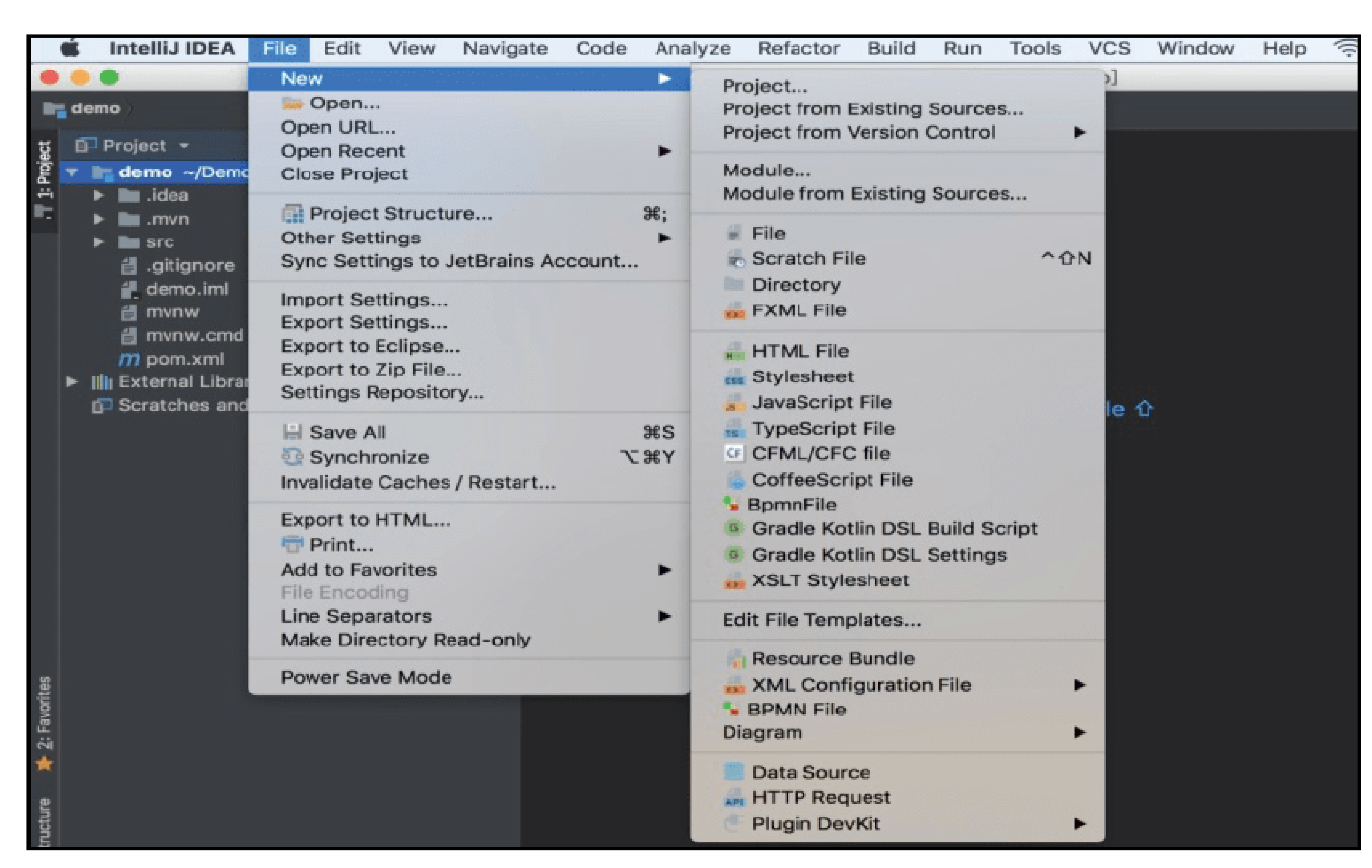


图 2-8 IntelliJ IDEA 创建项目界面

2.2.2　配置 JDK 版本和 Initializr Service URL

打开 New Project 界面后，在左侧选择 Spring Initializr，如图 2-9 所示。选择后右边会有两个选项，第一项是选择 JDK 版本，由于本书均采用 Spring Boot 2.0.3 版本，最低支持 JDK1.8，因此我们选择 JDK1.8。下面的 Initializr Service URL 是用来查询 Spring Boot 的当前版本和组件的网站。这两个选项配置完成之后，单击下方的 Next 按钮进入下一个步骤。

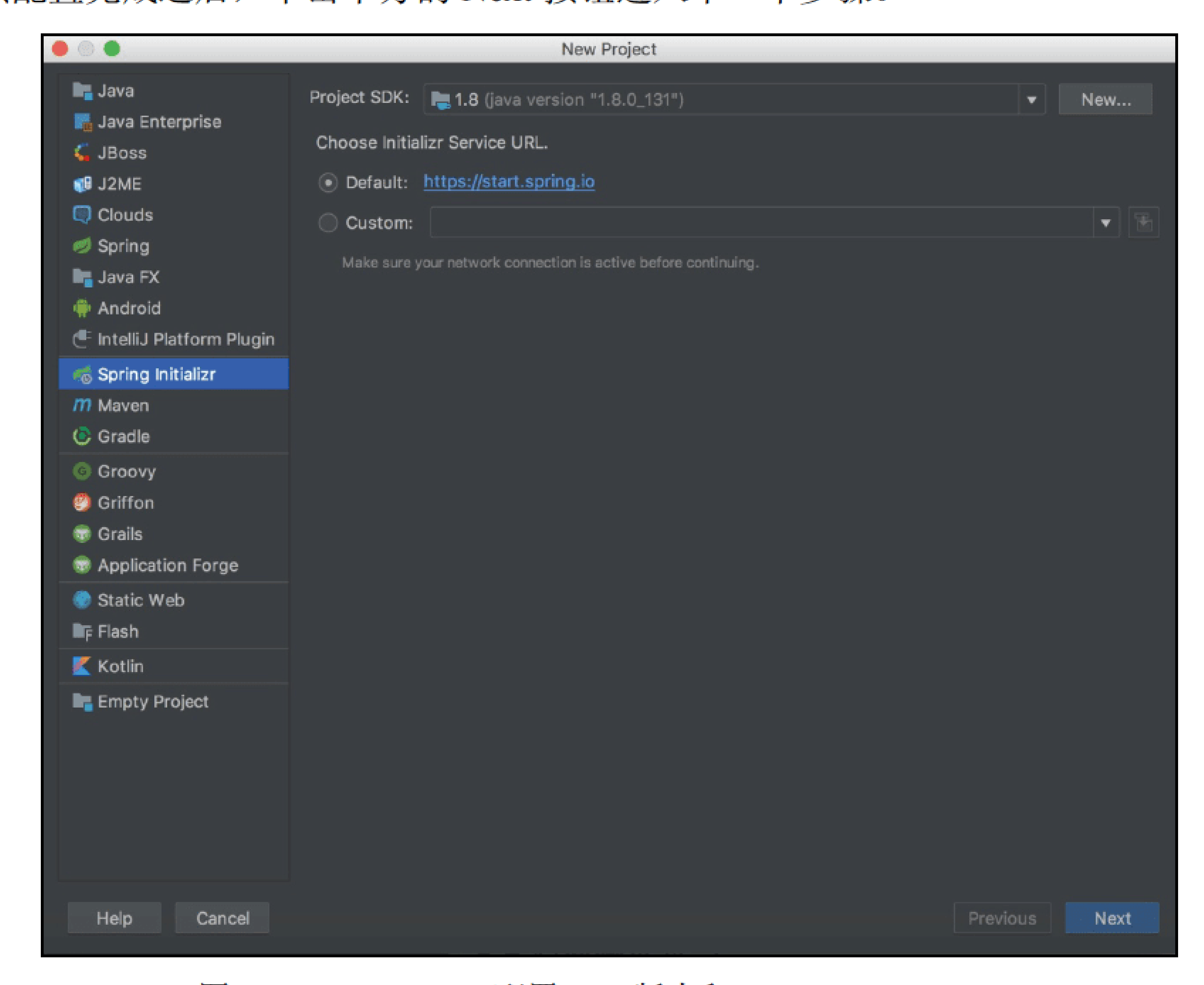


图 2-9　IntelliJ IDEA-配置 JDK 版本和 Initializr Service URL

2.2.3　配置 Project Metadata 信息

Project Metadata 配置信息包含如下内容，如图 2-10 所示。

- Group：项目组织的标识符。
- Artifact：项目标识符。
- Type：构建项目的方式，包含 Maven 和 Gradle，这里选择 Maven Project。
- Language：编程语言，这里选择 Java。
- Packaging：启动形式，包含 JAR 和 WAR，这里我们选择 JAR。
- Java Version：Java 版本。
- Version：项目版本号。
- Name：项目名称。
- Description：项目描述。
- Package：实际对应 Java 包的结构，是 main 目录里 Java 的目录结构。

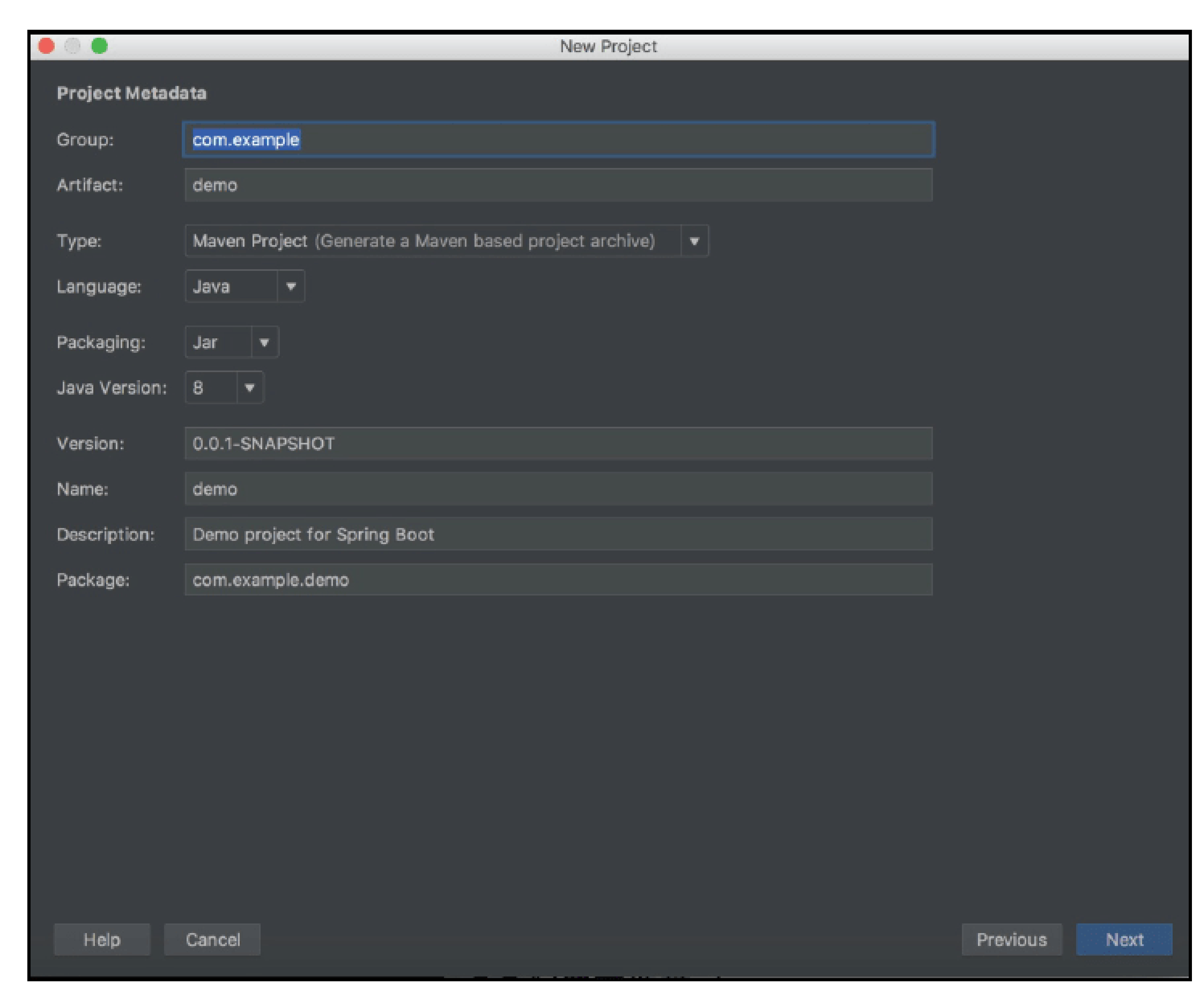


图 2-10　IntelliJ IDEA-配置 Project Metadata 信息

2.2.4　配置 Spring Boot 版本及默认引入组件

在 Spring Boot 下拉框中选择当前推荐的 Spring Boot 版本，在下方选择要使用的组件，然后单击 Next 按钮，如图 2-11 所示。

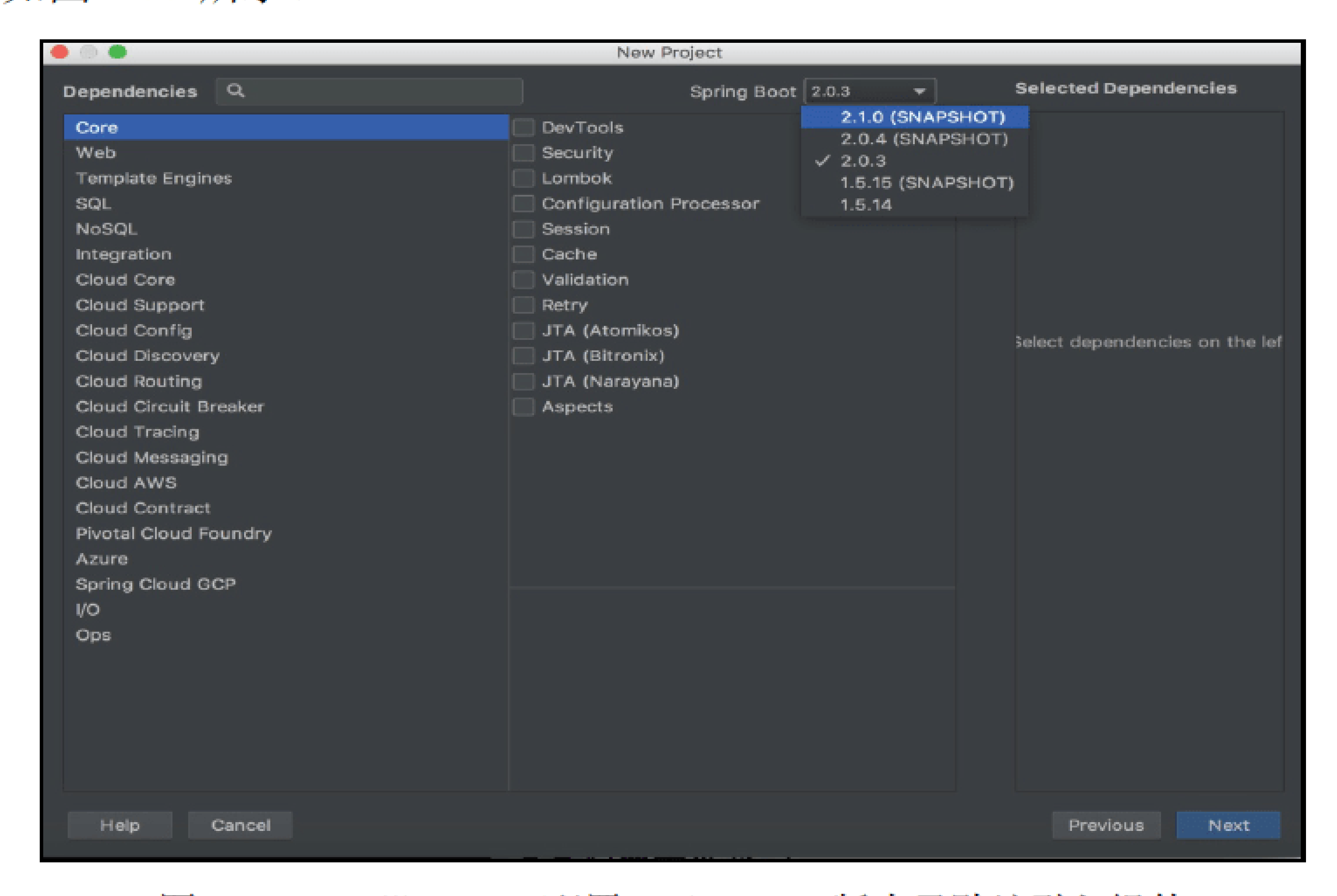


图 2-11　IntelliJ IDEA-配置 Spring Boot 版本及默认引入组件

2.2.5　配置项目名称和项目位置

在 Project Name 处配置项目名称，在 Project Location 处配置项目位置，如图 2-12 所示。配置完成后，单击 Finish 按钮即可完成项目的创建。



图 2-12　IntelliJ IDEA-配置项目名称和项目位置

到这里，我们已经完成了项目的创建。2.3 节将对项目的目录结构等进行进一步的介绍。

2.3　项目工程介绍

本节继续使用 2.2 节创建的项目。如图 2-13 所示。本节将介绍项目的工程目录结构，从图 2-13 中可以看到大致分为 4 部分。

（1）Java 类文件

（2）资源文件

（3）测试类文件

（4）pom 文件

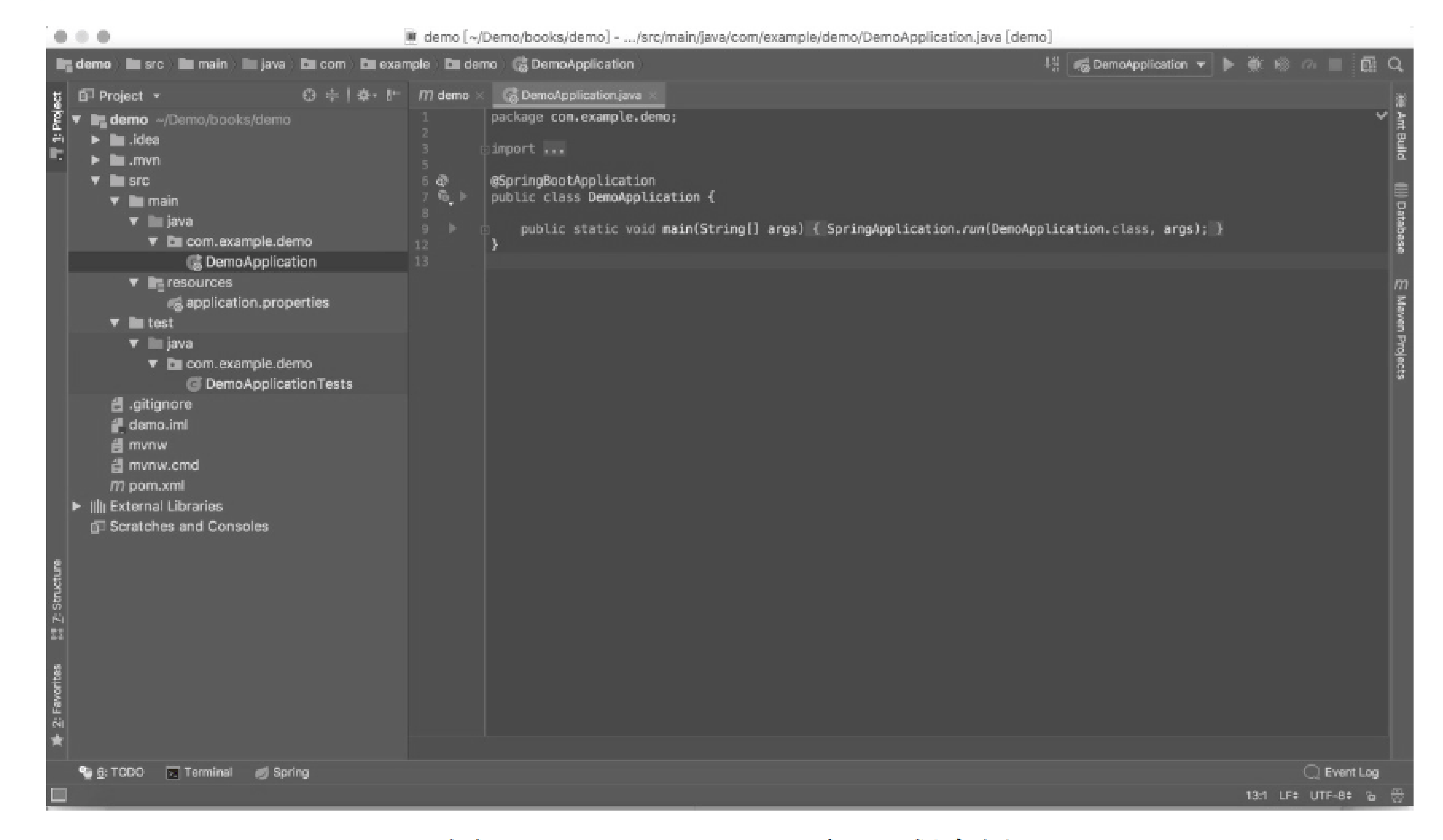


图 2-13　IntelliJ IDEA-项目工程介绍

2.3.1 Java 类文件

src/main/java 下用于放置 Java 类文件，由于这是一个新建的项目，因此目前只有一个 DemoApplication 类，如图 2-14 所示。这个类是 Spring Boot 应用的主程序，其中@SpringBootApplication 注解用来说明这是 Spring Boot 应用的启动类，其中包含自动配置、包扫描等功能，main 方法是启动应用的入口方法，命令行或者插件等任何方式启动，都会调用这个方法。

```
DemoApplication.java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) { SpringApplication.run(DemoApplication.class, args); }
}
```

图 2-14　IntelliJ IDEA-DemoApplication 类

2.3.2 资源文件

1. 配置文件

src/main/resources 下面主要用于放置 Spring Boot 应用的配置文件，新建项目的时候会默认创建一个 application.properties（默认是一个空文件），也可以将.properties 文件修改为.yml 文件，用缩进结构的键值对来进行配置。同时，配置文件可以进行一些应用需要的配置，如端口号等，后续章节会陆续介绍。

2. 静态资源

src/main/resources/static 下面主要放置应用的静态资源文件，如 HTML、JavaScript、图片等。

3. 模板文件

src/main/resources/templates 下面主要放置应用的模板文件，比如使用 Thymeleaf 后的 Thymeleaf 模板文件等。

2.3.3 测试类文件

src/test/java 下用于放置 Spring Boot 测试类文件，默认会根据项目名称创建一个测试类，如图 2-15 所示。打开该类可以发现@SpringBootTest 注解用于标明这是一个 Spring Boot 测试类。

```
DemoApplicationTests.java
package com.example.demo;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class DemoApplicationTests {

    @Test
    public void contextLoads() {
    }

}
```

图 2-15　IntelliJ IDEA-DemoApplicationTests 类

2.3.4　pom 文件

项目中还包含一个 pom.xml 文件，这是 Maven 项目用于构建项目的重要组成部分。从 pom 文件的完整代码中可以看到新建的 Spring Boot 项目默认的依赖以及版本号、Java 版本等，如代码清单 2-4 所示。

代码清单 2-4　Spring Boot 应用程序 pom 文件代码清单

```
    <?xml version="1.0" encoding="UTF-8"?>
    <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>com.example</groupId>
        <artifactId>demo</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>jar</packaging>

        <name>demo</name>
        <description>Demo project for Spring Boot</description>

        <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>2.0.3.RELEASE</version>
            <relativePath/> <!-- lookup parent from repository -->
        </parent>

        <properties>
            <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
            <project.reporting.outputEncoding>UTF-8
</project.reporting.outputEncoding>
            <java.version>1.8</java.version>
        </properties>

        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter</artifactId>
            </dependency>

            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
```

```
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

2.4 运 行 项 目

在此之前，我们已经简单地搭建了一个 Spring Boot 项目。接下来介绍在 IntelliJ IDEA 上如何运行项目。其实很简单，可以直接使用 IntelliJ IDEA 的 Run 或者 Debug 来启动，或者利用我们在 2.3 节介绍的 Spring Boot 主程序，直接运行主程序中的 main 函数来运行项目。无论采用哪一种，都可以启动项目。然后查看控制台，如图 2-16 所示。

图 2-16 Spring Boot 启动 Log

2.5 小 结

本章对 Spring Boot 开发环境进行了搭建，介绍了使用 IntelliJ IDEA 创建项目的方法和步骤，以及创建后的项目工程，最后介绍了如何在 IntelliJ IDEA 中运行项目。到这里可能还是有一些枯燥，但是当看到启动项目的 banner 时可能会焕然一新。截至目前，准备工作已经做好了，接下来将会进行真正的 Spring Boot 之旅。

第 3 章

Spring Boot 的 Web 之旅

在开发中，Web 项目与我们息息相关，本章将介绍 Spring Boot 的 Web 项目，从构建简单项目、使用模板框架、WebJars 等进行系统性的学习。

3.1　Spring Boot 的第一个 Web 项目

打开 IntelliJ IDEA，新建一个简单的项目，过程与第 2 章介绍的一致。

3.1.1　加入 Web 依赖

创建项目后，在项目的 pom 文件中加入 Web 依赖，并且导入依赖文件，如代码清单 3-1 所示。

代码清单 3-1　Spring Boot-Web 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

3.1.2　创建 Controller

新建一个 HelloController，在类上加入注解@RestController，了解 Spring MVC 的都知道，这个注解是 Spring 4.0 版本之后的一个注解，功能相当于@Controller 与@ResponseBody 两个注解的功能之和。

在 HelloController 内创建方法 hello()，在方法上加入注解@GetMapping("/hello")，这个注解是在 Spring 后期推出的一个组合注解，是@RequestMapping(method = RequestMethod.GET)的缩写，将 HTTP Get 映射到方法上。让 hello()返回一个字符串“Hello, This is your first Spring Boot Web Project !”。HelloController 的完整内容如代码清单 3-2 所示。

代码清单 3-2　HelloController 的完整内容

```
package com.springboot.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello(){
        return "Hello , This is your first SpringBoot Web Project !";
    }
}
```

3.1.3　测试运行

截至目前，其实简单的 Web 项目已经创建完成了，接下来启动项目。首先观察一下控制台，如图 3-1 所示，我们似乎得到几个信息：项目的端口是 8080、默认使用的 Web 容器是 Tomcat、刚刚写的 hello()在控制台有所映射。

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.3.RELEASE)

2018-07-29 15:39:07.662  INFO 25579 --- [           main] com.dalaoyang.Chapter31Application       : Starting Chapter31Application on dalaoyang-3.local with PID 25579
2018-07-29 15:39:07.666  INFO 25579 --- [           main] com.dalaoyang.Chapter31Application       : No active profile set, falling back to default profiles: default
2018-07-29 15:39:07.721  INFO 25579 --- [           main] ConfigServletWebServerApplicationContext : Refreshing org.springframework.boot.web.servlet.context.Annotatio
2018-07-29 15:39:08.991  INFO 25579 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2018-07-29 15:39:09.020  INFO 25579 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2018-07-29 15:39:09.021  INFO 25579 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.31
2018-07-29 15:39:09.026  INFO 25579 --- [ost-startStop-1] o.a.catalina.core.AprLifecycleListener   : The APR based Apache Tomcat Native library which allows optimal p
2018-07-29 15:39:09.172  INFO 25579 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2018-07-29 15:39:09.172  INFO 25579 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1455 ms
2018-07-29 15:39:09.467  INFO 25579 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Servlet dispatcherServlet mapped to [/]
2018-07-29 15:39:09.477  INFO 25579 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2018-07-29 15:39:09.478  INFO 25579 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2018-07-29 15:39:09.478  INFO 25579 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
2018-07-29 15:39:09.478  INFO 25579 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2018-07-29 15:39:09.836  INFO 25579 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org
2018-07-29 15:39:10.074  INFO 25579 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.web.servl
2018-07-29 15:39:10.220  INFO 25579 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/hello],methods=[GET]}" onto public java.lang.String co
2018-07-29 15:39:10.230  INFO 25579 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.Response
2018-07-29 15:39:10.234  INFO 25579 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springfr
2018-07-29 15:39:10.303  INFO 25579 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.spr
2018-07-29 15:39:10.304  INFO 25579 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframe
2018-07-29 15:39:10.503  INFO 25579 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2018-07-29 15:39:10.608  INFO 25579 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2018-07-29 15:39:10.614  INFO 25579 --- [           main] com.dalaoyang.Chapter31Application       : Started Chapter31Application in 3.335 seconds (JVM running for 4.
```

图 3-1　Spring Boot-Web 项目启动 Log

在浏览器上访问 http://localhost:8080/hello，可以看到浏览器打印了我们在方法内返回的内容。

```
Hello , This is you first SpringBoot Web Project !
```

到这里，一定会有人和笔者第一次接触的时候有同样的想法。Spring Boot 项目太神奇了，完全颠覆了我们对传统 Web 项目的认识，它没有原有的 web.xml 文件，只需短短的几行代码，就完成了原有 Spring MVC 项目的烦琐配置，甚至连配置 Tomcat 都不需要，直接在内部提供了 Tomcat。

3.2　WebFlux 的使用

前面介绍了 Spring Boot 使用 Spring MVC 模式搭建一个简单的 WebFlux 项目，本节为大家介绍 Spring Boot 提供的另一种模式—— Spring WebFlux。引用 Spring 官网的说明，我们在第 1 章已经看到过，如图 3-2 所示。

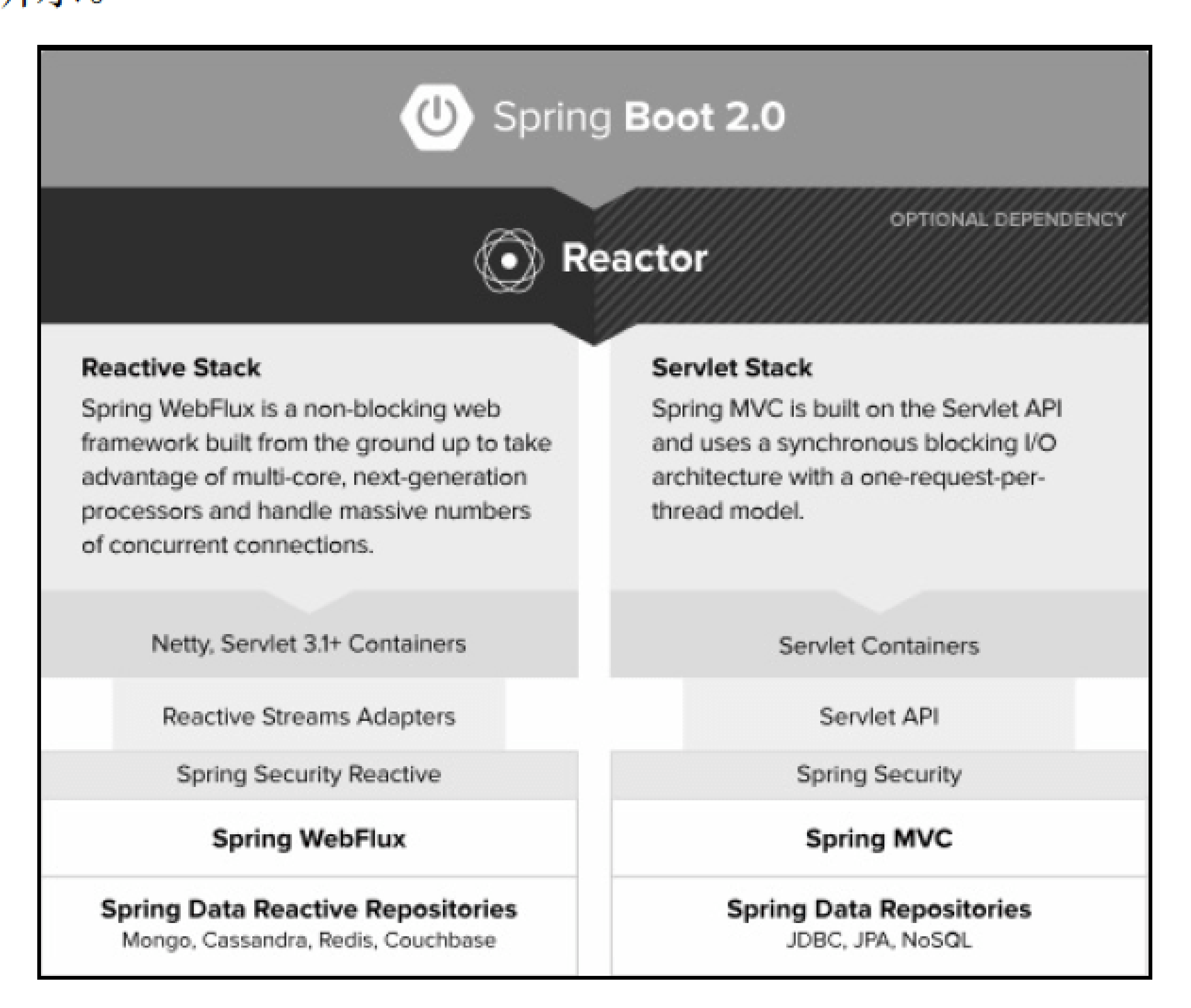


图 3-2　Spring MVC 与 Spring WebFlux 比较图

从图 3-2 可以看到，WebFlux 是一个非阻塞的 Web 框架，它不再完全依赖于 Servlet，而是实现了 Reactive Streams 规范。也就是说，可以使用响应式编程，但是并非无法运行在之前的 Servlet 容器上，只不过必须是在 Servlet 3.1 以上，并且默认推荐的是使用 Netty 这种异步容器。刚才我们提到了响应式编程，接下来利用响应式编程来创建一个 Spring Boot WebFlux 项目。

3.2.1　添加 WebFlux 依赖

首先创建一个项目，在项目的 pom 文件中添加 WebFlux 依赖。Spring WebFlux 同样支持传统 Spring MVC 使用注解的形式进行 WebFlux 跳转，同时支持函数式编程配置路由进行 WebFlux 跳转。传统模式就不再赘述了，这里以响应式编程为例，Spring WebFlux 依赖的内容如代码清单 3-3 所示。

代码清单 3-3　Spring Boot-WebFlux 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
```

3.2.2 创建一个处理方法类

新建类 HelloHandle，创建一个 hello 方法供接下来使用，其中返回值 Mono<ServerResponse> 作为响应对象，其中 ServerResponse 包含响应状态、响应头信息等，类上面的@Component 注解用于将类实例化到 Spring 容器中。总的来说，这个方法就是返回一句字符串，HelloHandle 类的内容如代码清单 3-4 所示。

代码清单 3-4　Spring Boot-WebFlux 项目 HelloHandle 的内容

```
    @Component
    public class HelloHandle {

        public Mono<ServerResponse> hello(ServerRequest request){
            return ServerResponse.ok().contentType(MediaType.APPLICATION_JSON)
                    .body(BodyInserters.fromObject("Hello, This is a Spring Boot
WebFlux Project !"));
        }

    }
```

3.2.3 创建一个 Router 类

创建一个 HelloRouter 类，用来定义路由信息，每个路由都会映射到对应的处理方法（功能类似于@RequestMapping）。当接收到对应 HTTP 请求后，调用此方法，通过 RouterFunctions.route（RequestPredicate, HandlerFunction）提供一个路由器函数的默认实现，HelloRouter 的内容如代码清单 3-5 所示。

代码清单 3-5　Spring Boot-WebFlux 项目 HelloRouter 的内容

```
    @Configuration
    public class HelloRouter {

        @Bean
        public RouterFunction<ServerResponse> routeHello(HelloHandle
helloHandle) {
            return RouterFunctions
                    .route(RequestPredicates.GET("/hello")
                    .and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
helloHandle::hello);
        }
    }
```

3.2.4　测试运行

启动项目，我们来观察一下控制台，如图 3-3 所示。可以从第 4 行看到，刚刚写的 hello 映射已经成功了。正如之前介绍的，默认启用的 Netty 容器运行端口默认为 8080。

```
 :: Spring Boot ::        (v2.0.3.RELEASE)

2018-08-11 15:36:07.837  INFO 3710 --- [           main] com.dalaoyang.Chapter32Application       : Starting Chapter32Application on dalaoyang-3.local with PID 3710 (
2018-08-11 15:36:07.842  INFO 3710 --- [           main] com.dalaoyang.Chapter32Application       : No active profile set, falling back to default profiles: default
2018-08-11 15:36:07.922  INFO 3710 --- [           main] onfigReactiveWebServerApplicationContext : Refreshing org.springframework.boot.web.reactive.context.Annotatio
2018-08-11 15:36:09.075  INFO 3710 --- [           main] o.s.w.r.f.s.s.RouterFunctionMapping      : Mapped ((GET && /hello) && Accept: [application/json]) -> com.dala
2018-08-11 15:36:09.094  INFO 3710 --- [           main] o.s.w.r.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.spri
2018-08-11 15:36:09.094  INFO 3710 --- [           main] o.s.w.r.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframew
2018-08-11 15:36:09.189  INFO 3710 --- [           main] o.s.w.r.r.m.a.ControllerMethodResolver   : Looking for @ControllerAdvice: org.springframework.boot.web.reacti
2018-08-11 15:36:09.589  INFO 3710 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2018-08-11 15:36:09.709  INFO 3710 --- [ctor-http-nio-1] r.ipc.netty.tcp.BlockingNettyContext     : Started HttpServer on /0:0:0:0:0:0:0:0:8080
2018-08-11 15:36:09.710  INFO 3710 --- [           main] o.s.b.web.embedded.netty.NettyWebServer  : Netty started on port(s): 8080
2018-08-11 15:36:09.717  INFO 3710 --- [           main] com.dalaoyang.Chapter32Application       : Started Chapter32Application in 2.348 seconds (JVM running for 3.8
2018-08-11 15:37:42.521  WARN 3710 --- [ctor-http-nio-2] .a.w.r.e.DefaultErrorWebExceptionHandler : Failed to handle request [GET http://localhost:8080/favicon.ico]:
```

图 3-3　Spring Boot-WebFlux 项目启动 Log

在浏览器上访问 http://localhost:8080/hello 可以看到：

```
Hello, This is a Spring Boot WebFlux Project !
```

响应式编程的简单实现到这里就结束了，可能在工作和学习上两种方式有不同的使用情况，无论是响应式编程还是非响应式编程，都有各自不同的好处，这里不做更多的比较了，具体可以按照自己的实际需求来选择。

3.3　使用热部署

热部署这个词汇大家听起来应该并不陌生，在 Spring Boot 框架中是否提供了相关的热部署呢？其实在第 1 章介绍 Spring Boot 框架的特点时已经指出了，只需要引入 spring-boot-devtools 依赖文件即可，十分简单。引入依赖后，重新编译修改的类文件或配置文件等（笔者默认快捷键是 Command+F9），Spring Boot 框架会自动替我们重启，spring-boot-devtools 依赖如代码清单 3-6 所示。

代码清单 3-6　spring-boot-devtools 依赖内容

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

3.4 配 置 文 件

在第 2 章介绍 Spring Boot 项目结构的时候简单提到了配置文件，本节将对 Spring Boot 的配置文件进行介绍。

3.4.1 配置文件类型

在使用 IntelliJ IDEA 创建 Spring Boot 项目时，IDE 会在 src/main/java/resources 目录下创建一个 application.properties 文件。在这种情况下，我们使用配置的时候需要使用下面的格式（以端口号配置为例），如代码清单 3-7 所示。

代码清单 3-7 properties 文件配置端口号

```
server.port=8888
```

当然，我们也可以将配置文件 application.properties 后缀修改为.yml 格式，即文件全名为 application.yml。在这种格式下，端口配置如代码清单 3-8 所示。

代码清单 3-8 YML 文件配置端口号

```
server:
   port:8888
```

3.4.2 自定义属性

前面介绍了两种配置文件的格式，这里以 properties 文件为例，在 application.properties 中自定义几个属性，如代码清单 3-9 所示。

代码清单 3-9 Spring Boot 配置自定属性

```
book.name=Spring Boot 2 实战之旅
book.author=杨洋
```

在类中，如果需要读取配置文件的内容，那么只需要在属性上使用@Value("${属性名}")，新建一个 TestController，在其中创建一个 test1 方法进行测试。TestController 的完整内容如代码清单 3-10 所示。

代码清单 3-10 TestController 类的完整内容

```
@RestController
public class TestController {
```

```
    @Value("${book.name}")
    private String bookName;

    @Value("${book.author}")
    private String bookAuthor;

    @GetMapping("test1")
    public String test1(){
       return "本书书名是：" + bookName + ",作者是：" + bookAuthor;
    }
}
```

启动工程，在浏览器上访问 http://localhost:8080/test1，可以看到浏览器显示：“本书书名是：Spring Boot 2 实战之旅,作者是：杨洋”。

注　意

在 application.properties 中配置中文值，读取时会出现中文乱码问题。因为 Java 默认会使用 ISO-8859-1 的编码方式来读取*.properties 配置文件，而 SpringBoot 应用则以 UTF-8 的编码方式来读取，就导致产生了乱码问题。

对于这个问题，官方推荐的做法是：“Characters that cannot be directly represented in this encoding can be written using Unicode escapes”，大致意思就是使用 Unicode 的方式来展示字符。例如上述代码中的 book.author=杨洋应该配置成 book.author=\u6768\u6d0b。

3.4.3　使用随机数

在配置文件中，还提供了随机数供我们使用，即在配置文件中使用${random}来生成不同类型的随机数，大致分为随机数、随机 uuid、随机字符串等。在配置文件内添加几种利用随机数创建的属性，如代码清单 3-11 所示。

代码清单 3-11　配置文件使用随机数

```
# 随机字符串
book.value=${random.value}
# 随机 int 值
book.intValue=${random.int}
# 随机 long 值
book.longValue=${random.long}
# 随机 uuid
book.uuid=${random.uuid}
```

```
# 1000 以内随机数
book.randomNumber=${random.int(1000)}
# 自定义属性间引用
book.title=书名是： ${book.name}
```

在配置了这么多属性后，可以使用 JavaBean 模式来给属性赋值，创建一个 BookConfigBean 实体类。由于自定义属性的前缀都是由 book 开头的，因此我们可以在实体类上加入注解 @ConfigurationProperties(prefix = "book")，同时需要在启动类上加入注解 @EnableConfigurationProperties(BookConfigBean.class)，表明启动这个配置类。实体类内容如代码清单 3-12 所示（这里省略了 set、get 方法）。

代码清单 3-12　BookConfigBean 类的完整内容

```
@ConfigurationProperties(prefix = "book")
public class BookConfigBean {
    private String name;
    private String author;
    private String value;
    private int intValue;
    private long longValue;
    private String uuid;
    private int randomNumber;
    private String title;
}
```

到这里，配置就完成了。接下来在 TestController 中利用@Autowired 注解注入 BookConfigBean 类，并且创建一个 test2 方法进行测试。test2 方法及注入 BookConfigBean 类的内容如代码清单 3-13 所示。

代码清单 3-13　TestController 类新增内容

```
@Autowired
private BookConfigBean bookConfigBean;

@GetMapping("test2")
public BookConfigBean test2(){
    return bookConfigBean;
}
```

在浏览器上访问 http://localhost:8080/test2 进行测试，显示结果如下：

```
{"name":"Spring Boot 2 实战之旅","author":"杨洋","value":
"014ccb5f689f3c2b528a32cd755d3921","intValue":-719017145,"longValue":78148173
79928523304,"uuid":"a800992e-8262-426d-8aab-66aae9df798a","randomNumber":918,
"title":"本书书名是： Spring Boot 2 实战之旅"}
```

3.4.4　多环境配置

在开发 Spring Boot 项目的时候，可能有这样的情况，一套程序需要在不同的环境中发布，数据库配置、端口配置或者其他配置各不相同，如果每次都需要修改为对应环境配置，不仅耗费人力，而且特别容易出现错误，造成不必要的麻烦。

通常情况下，我们可以配置多个配置文件，在不同的情况下进行替换。而在 Spring Boot 项目中，我们新建几个配置文件，文件名以 application-{name}.properties 的格式，其中的{name}对应环境标识，比如：

- application-dev.properties：开发环境。
- application-test.properties：测试环境。
- application-prod.properties：生产环境。

然后，可以在主配置文件（application.properties）中配置 spring.profiles.active 来设置当前要使用的配置文件。比如，在主配置文件中配置本次指定使用的配置文件后缀，配置内容如代码清单 3-14 所示。

代码清单 3-14　application.properties 配置文件

```
spring.profiles.active=test
```

创建 application-dev.properties 配置文件，在文件中配置端口号为 8081，配置文件内容如代码清单 3-15 所示。

代码清单 3-15　application-dev.properties 配置文件

```
server.port=8081
```

创建 application-test.properties 配置文件，在文件中配置端口号为 8082，配置文件内容如代码清单 3-16 所示。

代码清单 3-16　application-test.properties 配置文件

```
server.port=8082
```

启动项目或者打成 JAR 包形式都会自动读取对应配置文件，可以在控制台看到启动端口号为 8082。

3.4.5　自定义配置文件

前面介绍了多环境配置文件，我们也可以使用自定义配置文件，比如新建一个 test.properties，配置文件内容如代码清单 3-17 所示。

代码清单 3-17 test.properties 配置文件

```
com.book.name=Spring Boot 2 实战之旅
com.book.author=杨洋
```

与之前一样，新建一个 javabean 来读取配置文件。新建一个 ConfigBean，在类上加上注解 @PropertySource(value = "classpath:test.properties")，并且和之前一样需要加入@ConfigurationProperties (prefix = "com.book")，实体类代码如代码清单 3-18 所示（省略了 set、get 方法）。

代码清单 3-18 ConfigBean 类的完整内容

```
@Component
@PropertySource(value = "classpath:test.properties")
@ConfigurationProperties(prefix = "com.book")
public class ConfigBean {
    private String name;
    private String author;
}
```

同样，在 TestController 中注入 bean 并且创建测试方法，内容如代码清单 3-19 所示。

代码清单 3-19 TestController 类新增内容

```
@Autowired
private ConfigBean configBean;

@GetMapping("test3")
public ConfigBean test3(){
    return configBean;
}
```

使用浏览器访问 http://localhost:8080/test3，可以看到显示如下内容：

```
{"name":"Spring Boot 2 实战之旅","author":"杨洋"}
```

3.5 使用页面模板

在 Web 开发过程中，前后端交互是一件不可避免的事情。接下来我们学习 Spring Boot 常用的页面模板框架。

3.5.1 使用 Thymeleaf

Thymeleaf 是当今比较流行的模板框架，并且是 Spring Boot 官方推荐使用的模板框架。本小

节介绍 Spring Boot 框架如何使用 Thymeleaf，并且会对 Thymeleaf 框架的使用方法进行介绍。

首先创建项目，在项目中加入 spring-boot-starter-thymeleaf 依赖。这里需要提醒的是，由于 Thymeleaf 对 HTML 的校验特别严格，比如标签没有结束等可能会对不熟悉者造成未知的困惑，因此我们还需要加入 nekohtml 的依赖来避免这个“坑”。Thymeleaf 依赖如代码清单 3-20 所示。

代码清单 3-20　Thymeleaf 项目-pom 文件内容

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!--去除 html 严格校验 -->
<dependency>
     <groupId>net.sourceforge.nekohtml</groupId>
     <artifactId>nekohtml</artifactId>
     <version>1.9.22</version>
</dependency>
```

完成依赖的配置之后，我们需要在配置文件中对 Thymeleaf 进行配置，比如编码格式、缓存设置、文件前后缀等。配置文件内容如代码清单 3-21 所示。

代码清单 3-21　Thymeleaf 项目-配置文件内容

```
## thymeleaf 缓存是否开启，开发时建议关闭，否则更改页面后不会实时展示效果
spring.thymeleaf.cache=false
## thymeleaf 编码格式
spring.thymeleaf.encoding=UTF-8
## thymeleaf 对 HTML 的校验很严格，用这个去除 thymeleaf 严格校验
spring.thymeleaf.mode=LEGACYHTML5
## thymeleaf 模板文件前缀
spring.thymeleaf.prefix=classpath:/templates/
## thymeleaf 模板文件后缀
spring.thymeleaf.suffix=.html
```

到这里，准备工作已经完成。需要做的是创建一个 Controller 和 HTML 进行测试。新建一个 IndexController，我们先写一个简单的路由跳转方法并且传一个字符串值进行测试。IndexController 内容如代码清单 3-22 所示。

代码清单 3-22　Thymeleaf 项目-IndexController 的内容

```
@Controller
public class IndexController {

    @GetMapping("/")
    public String index(ModelMap modelMap){
        modelMap.addAttribute("msg", "Hello , Dalaoyang !");
        return "index";
    }
}
```

然后，在 src/mian/resources/templates 下新建一个 index.html（需要结合配置文件中 spring.thymeleaf.prefix 的配置信息存放 HTML），使用 th:text="${msg}"来接收后台传来的数据。index.html 内容如代码清单 3-23 所示。

代码清单 3-23　Thymeleaf 项目-index.html 的内容

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1 th:text="${msg}"></h1>
</body>
</html>
```

启动项目，在浏览器上访问 http://localhost:8080，可以看到有如下显示：

```
Hello, Dalaoyang !
```

其实到这里 Spring Boot 整合 Thymeleaf 已经完成，但是为了方便后面章节的使用，笔者在这里再介绍一下 Thymeleaf 模板的常用语法。

- th:text　设置当前元素的文本内容。
- th:value　设置当前元素的值。
- th:each　循环遍历元素，一般配合上面两者使用。
- th:attr　设置当前元素的属性。
- th:if th:switch th:case th:unless　用作条件判断。
- th:insert th:replace th:incloud　代码块引入，一般用作提取公共文件，或者引用公共静态文件等。

当然，Thymeleaf 也提供了一些内置方法供我们使用，比如：

- #numbers　数字方法。

- #dates 日期方法。
- #calendars 日历方法。
- #strings 字符串方法。
- #lists 集合方法。
- #maps 对象方法。

关于 Thymeleaf 先了解到这里，后面的章节会对它有具体的实战使用，这里就不再赘述了。

3.5.2 使用 FreeMarker

刚刚介绍了 Thymeleaf 模板，接下来我们学习 FreeMarker 模板，无论是语法还是配置等，两者都有很多相似的地方。接下来，我们学习 Spring Boot 项目整合 FreeMarker 模板。

新建项目，在项目中加入 Freemarker 依赖，如代码清单 3-24 所示。

代码清单 3-24 FreeMarker 项目-pom 文件内容

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-freemarker</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

接下来配置 FreeMarker 模板属性，与 Thymeleaf 模板配置类似，唯一需要注意的是模板文件后缀配置的是 FTL 文件。配置文件如代码清单 3-25 所示。

代码清单 3-25 FreeMarker 项目-配置文件内容

```
## freemarker 缓存是否开启
spring.freemarker.cache=false
## freemarker 编码格式
spring.freemarker.charset=UTF-8
## freemarker 模板文件前缀
spring.freemarker.template-loader-path=classpath:/templates/
## freemarker 模板文件后缀，注意这里后缀名是.ftl
spring.freemarker.suffix=.ftl
```

接下来，创建一个 IndexController 进行测试，内容如代码清单 3-26 所示。

代码清单 3-26　FreeMarker 项目-IndexController 文件内容

```
@Controller
public class IndexController {

    @GetMapping("/")
    public String index(ModelMap modelMap) {
        modelMap.addAttribute("msg", "Hi , Dalaoyang !");
        return "index";
    }
}
```

在 src/resources/templates 下新建 index.ftl（注意文件后缀），使用${msg}接收后来传送的数据，文件内容如代码清单 3-27 所示。

代码清单 3-27　FreeMarker 项目-index.ftl 文件内容

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>${msg}</h1>
</body>
</html>
```

到这里，项目配置完成。启动项目，在浏览器上访问 http://localhost:8080，可以看到如下结果：

```
Hi, Dalaoyang !
```

接下来介绍 FreeMarker 的常用语法。

（1）通用赋值：${xxx}格式

- 比如后台返回键值 aaa=string，可以使用${aaa?string}，输出“Hi , Dalaoyang !”。
- 比如后台返回键值 aaa="2018-08-01 23:59"，可以使用${aaa?string("EEE,MMM d,yy")}，输出：星期二，八月 14,18。
- 比如后台返回键值 aaa=false，可以使用${aaa?string("是","否")}，输出：否。

（2）数值赋值：#{xxx}或者#{xxx;format} 格式

后者 format 可以是以下格式（其中 X 和 Y 为数字）：

- mX　小数部分最小 X 位，比如后台返回值 aaa=3.782131，可以使用#{x;m2}，输出 3.78。
- MX　小数部分最大 X 位，比如后台返回值 aaa=3.782131，可以使用#{x;M3}，输出 3.782。

- mXMY　小数部分最小 X 位，最大 Y 位，比如后台返回值 aaa=3.782131，可以使用#{x;m1M3}，输出 3.782。

（3）常用内建函数

- html　对字符串进行 HTML 编码。
- lower_case　字符串转小写。
- upper_case　字符串转大写。
- trim　去前后空格。
- size　获取集合元素数量。
- int　获取数字部分。

（4）常用指令

- if elseif else　分支控制语句。
- list　输出集合数据。
- import　导入变量。
- include　类似于包含指令。

关于 FreeMarker 模板的内容到这里暂时结束了，毕竟这是一本关于 Spring Boot 的书，详细内容可以参考官方文档进行系统学习。

3.5.3　使用传统 JSP

虽然 Spring Boot 不建议使用 JSP 作为渲染页面，但是一定要使用的话，也是可以的。

新建项目，加入 JSP 对应的依赖和 JSTL 表达式依赖，并且需要注意 packaging 内不是 JAR 而是 WAR。pom 文件代码如代码清单 3-28 所示。

代码清单 3-28　JSP 项目-pom 文件内容

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>org.apache.tomcat.embed</groupId>
   <artifactId>tomcat-embed-jasper</artifactId>
   <scope>provided</scope>
</dependency>
```

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

然后，需要在 src/mian 下新建一个 webapp 目录，并且在其下新建 WEB-INF/jsp 文件夹，用于放置 JSP 页面，结构图如图 3-4 所示。

接下来，我们进行配置文件的配置，主要配置 JSP 页面文件前缀和后缀，基本上和 Thymeleaf、FreeMarker 类似，配置如代码清单 3-29 所示。

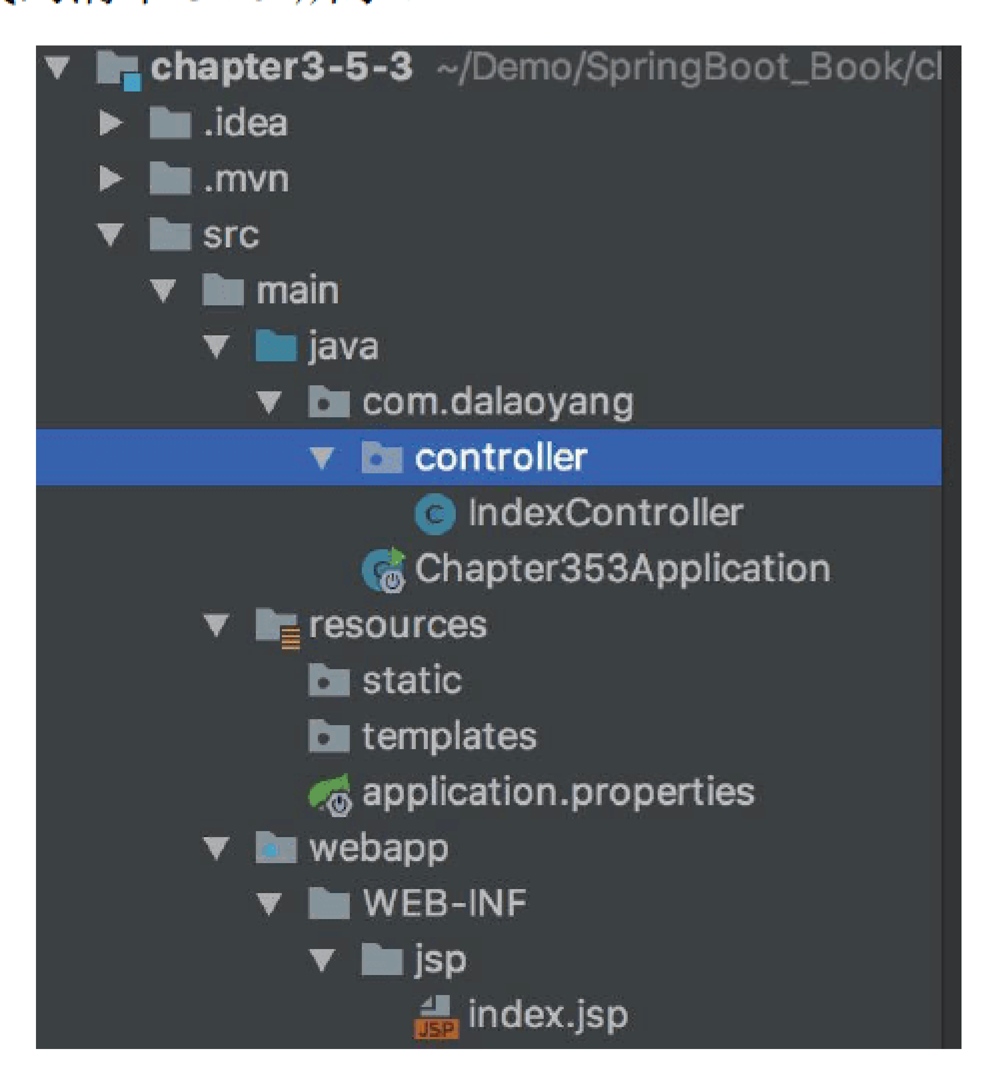


图 3-4 JSP 项目启动 Log

代码清单 3-29 JSP 项目-配置文件内容

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

然后创建一个 IndexController 文件作为跳转，完整内容如代码清单 3-30 所示。

代码清单 3-30 JSP 项目-IndexController 文件内容

```
@Controller
public class IndexController {
    @GetMapping("/")
    public String index(Model model){
        model.addAttribute("name", "dalaoyang");
        return "index";
    }
}
```

最后，在创建的 JSP 存放文件夹下创建一个 index.jsp，其中${name}用于接收后台传来的值。JSP 页面代码如代码清单 3-31 所示。

代码清单 3-31　JSP 项目-index.jsp 文件内容

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello</title>
</head>
<body>
Hello,${name}
</body>
</html>
```

在项目目录下使用命令 mvn spring-boot:run 启动项目，在浏览器上访问 http://localhost:8888/，可以看到如下结果：

```
Hello,dalaoyang
```

到这里，Spring Boot 使用 JSP 介绍完了。对于 Spring Boot 还有很多模板框架可以使用，如果不是必需的，那么建议不要使用。

3.6　使用 WebJars

在开发的过程中，很多时候需要结合前端进行开发。本节将介绍 Spring Boot 框架整合 WebJars 进行前端静态 JavaScript 和 CSS。

作为开发者，对 Bootstrap 和 jQuery 应该不会陌生。接下来我们将在 Spring Boot 项目中引入 WebJars，对应二者的 JAR 进行使用，在 pom 文件中加入二者的依赖文件，如代码清单 3-32 所示。

代码清单 3-32　WebJars 项目-pom 文件内容

```
<!-- 引用 bootstrap -->
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.3.7-1</version>
</dependency>
<!-- 引用 jquery -->
<dependency>
    <groupId>org.webjars</groupId>
```

```
        <artifactId>jquery</artifactId>
        <version>3.1.1</version>
    </dependency>
```

其实到这里整合完毕了，但是为了证实我们是否可以成功引用，在 src/main/recources/static 文件夹下新建 index.html，在 HTML 中引入刚刚加入依赖的文件。index.html 页面代码如代码清单 3-33 所示。

代码清单 3-33　WebJars 项目-index.html 文件内容

```
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Dalaoyang</title>
        <link rel="stylesheet" href="/webjars/bootstrap/3.3.7-1/css/
bootstrap.min.css" />
        <script src="/webjars/jquery/3.1.1/jquery.min.js"></script>
        <script src="/webjars/bootstrap/3.3.7-1/js/bootstrap.min.js">
        </script>

    </head>
    <body>
    <div class="container"><br/>
        <div class="alert alert-success">
            <a href="#" class="close" data-dismiss="alert" aria-label="close"> ×
</a>
            Hello, <strong>Dalaoyang!</strong>
        </div>
    </div>
    </body>
    <script type="text/javascript">
        alert($('.close').attr('href'));
    </script>
    </html>
```

在 HTML 页面中，我们分别对 Bootstrap 和 jQuery 进行了引用，使用 Bootstrap 对 a 标签进行了样式的修饰，使用 jQuery 在打开页面时利用告警输出了 a 标签的 href 值。启动项目，让我们来证实一下，在浏览器上访问 http://localhost:8080，如图 3-5 所示。

如图 3-5 所示，可以看到之前的操作都实现了。其实 WebJars 还提供了很多其他的依赖，具体使用可以查看 WebJars 官网（官网地址：https://www.webjars.org/）。



图 3-5　WebJars 项目访问效果图

3.7　国际化使用

对于很多门户网站，可能有很多客户来源于其他国家，这时就需要使用国际化来进行对外的交流。那么，在 Spring Boot 项目中是如何使用国际化的呢？

接下来使用一个小例子介绍 Spring Boot 项目如何运用国际化。

本节使用的依赖文件与 3.5 节使用 Thymeleaf 所使用的依赖文件以及配置文件完全一致，这里不再展示。

Spring Boot 在默认情况下是支持国际化使用的，首先需要在 src/main/resources 下新建国际化资源文件，这里为了举例说明，分别创建如下三个文件：

- messages.properties（默认配置），内容如代码清单 3-34 所示。

代码清单 3-34　国际化项目-默认语言配置文件内容

```
message = 欢迎使用国际化（默认）
```

- messages_en_US.properties（英文配置），内容如代码清单 3-35 所示。

代码清单 3-35　国际化项目-英文配置文件内容

```
message = Welcome to internationalization (English)
```

- messages_zh_CN.properties（汉语配置），内容如代码清单 3-36 所示。

代码清单 3-36　国际化项目-汉语配置文件内容

```
message = \u6b22\u8fce\u4f7f\u7528\u56fd\u9645\u5316\uff08\u4e2d\u6587\
uff09
```

然后就到了国际化的重头戏，需要进行 i18n 的配置，这里新建配置类 i18nConfig，这个类需要继承 WebMvcConfigurerAdapter 类。其中，在 localeResolver()方法中设置默认使用的语言类型，在 localeChangeInterceptor()方法中设置识别语言类型的参数，并且从继承类中实现 addInterceptors()方法，用于拦截 localeChangeInterceptor()方法，进而实现国际化。i18nConfig 类代码如代码清单 3-37 所示。

代码清单 3-37　国际化项目-i18nConfig 文件内容

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class i18nConfig  extends WebMvcConfigurerAdapter{
    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver slr = new SessionLocaleResolver();
        // 默认使用的语言
        slr.setDefaultLocale(Locale.US);
        return slr;
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
        // 参数名，用于区别使用的语言类型
        lci.setParamName("lang");
        return lci;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
}
```

改造默认生成的启动类，在类上加入 SpringMVC 注解@Controller，注入 MessageSource 类获取国际化资源，并且创建方法返回资源文件对应的数据，返回到前台。新增代码如代码清单 3-38 所示。

代码清单 3-38　国际化项目-启动类新增代码文件内容

```
@GetMapping("/")
public String hello(Model model){
    Locale locale = LocaleContextHolder.getLocale();
```

```
        model.addAttribute("message", messageSource.getMessage("message", null,
locale));
    return "index";
    }
```

在 src/main/resources/template 下新建 index.html，在页面中创建两个按钮，单击按钮切换语言。index.html 页面代码如代码清单 3-39 所示。

代码清单 3-39　国际化项目-index.html 文件内容

```
    <!DOCTYPE html>
    <html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
    <head>
        <meta charset="UTF-8">
        <title>Title</title>
    </head>
    <body>
    <a href="/?lang=en_US">English(US)</a>
    <a href="/?lang=zh_CN">简体中文</a></br>
    <p><label th:text="#{message}"></label></p>
    </body>
    </html>
```

启动项目，在浏览器上访问 http://localhost:8080/，显示的内容如图 3-6 所示。

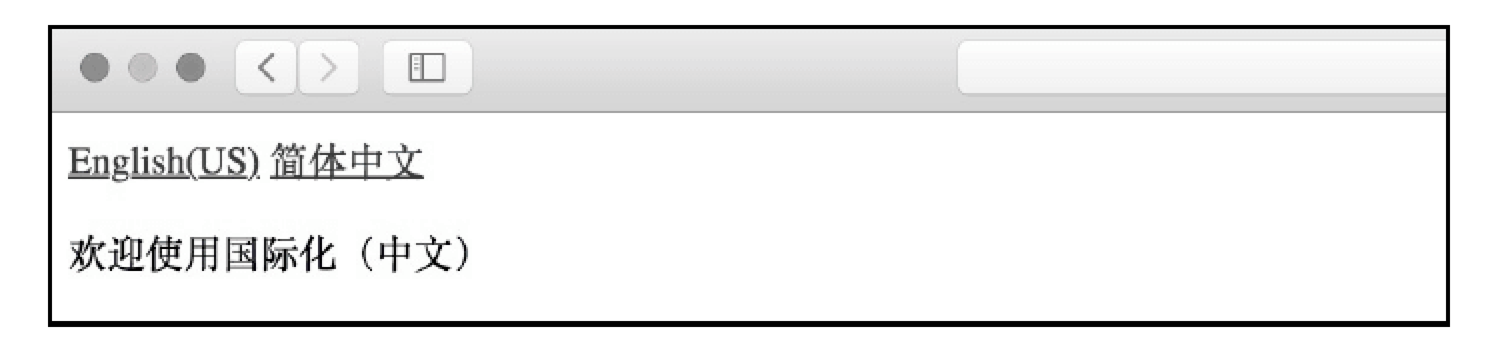


图 3-6　国际化项目，中文显示效果

单击页面中的 English(US)英文按钮，显示的内容如图 3-7 所示。

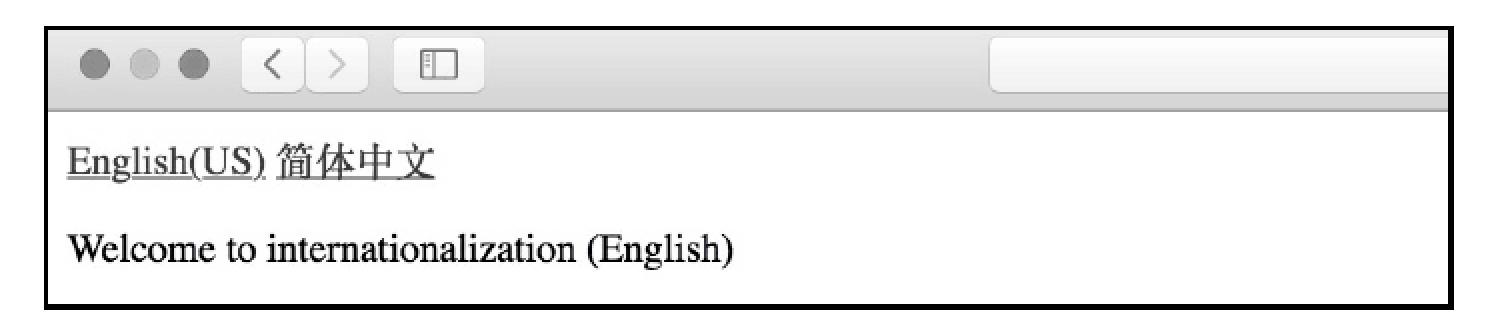


图 3-7　国际化项目，英文显示效果

这时你可能会有一个疑问，为什么没有显示默认的配置文件？这是因为在发送 HTTP 请求的时候，浏览器会根据你的请求头判断区域而进行系统设定。那么问题来了，怎么才会使用到默认的配置文件呢？其实很简单，浏览器根据系统区域在你的程序中找不到语言时，就会使用默认配置，比如，删除项目中英文和中文的配置，只留下一个默认配置，重启项目，再次访问 http://localhost:8080/，显示的内容如图 3-8 所示。

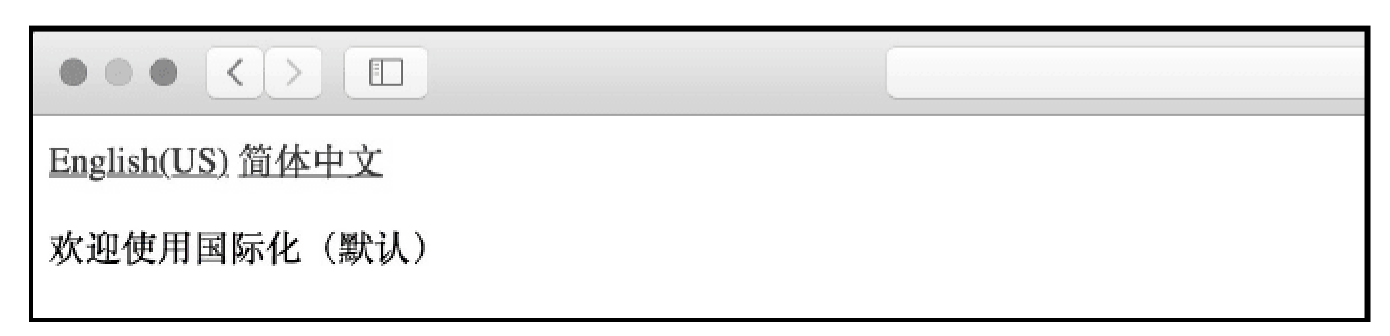


图 3-8 国际化项目，默认显示效果

这时就可以看到默认配置，而且即使你单击上面的两个切换语言的按钮，也不会有所改变，因为应用内现在只有这一种配置。

3.8 文件的上传和下载

3.7 节介绍了利用 Thymeleaf 模板进行国际化的使用，本节将使用 FreeMarker 模板进行文件的上传和下载，对前面 Spring Boot 使用模板框架进行一个回顾。

创建项目、项目依赖和配置文件与 3.5 节使用 FreeMarker 一致。在配置完依赖后，在 src/main/resources/templates 下新建一个 index.ftl 文件，文件内分别利用表单提交的方式写了两个表单，用于单个上传和批量上传，并且使用超链接的方式提供了一个下载方法，代码如代码清单 3-40 所示。

代码清单 3-40 上传和下载项目-index.flt 文件内容

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>${msg}</title>
</head>
<body>
<p>单文件上传</p>
<form action="upload" method="POST" enctype="multipart/form-data">
    文件：<input type="file" name="file"/>
    <input type="submit"/>
</form>
<hr/>
<p>文件下载</p>
<a href="download">下载文件</a>
<hr/>
<p>多文件上传</p>
<form  action="batch" method="POST" enctype="multipart/form-data">
    <p>文件 1：<input type="file" name="file"/></p>
    <p>文件 2：<input type="file" name="file"/></p>
    <p><input type="submit" value="上传"/></p>
</form>
```

```
</body>
</html>
```

更改启动类，在类上添加注解@Controller，新建 index 方法用于跳转，向页面传值 msg，方法如代码清单 3-41 所示。

代码清单 3-41　上传和下载项目-启动类新增文件内容

```
@GetMapping("/")
public String index(ModelMap modelMap){
    modelMap.addAttribute("msg", "文件上传下载");
    return "index";
}
```

接下来创建一个 FileController 用于文件上传和下载测试，具体方法如下：

（1）单个上传方法。可以根据页面上使用的 input 标签的 name 值获取对应内容，因为是文件，所以可以使用 MultipartFile 对象来接收文件，由于只是简单测试，因此利用 File 类自带的 transferTo 方法直接将文件存入对应存储位置。

（2）批量上传方法。获取页面内容的方式和单个上传方法大致相同，不同的是取得文件后，这里使用 BufferedOutputStream 流来进行上传，如果对 Java 流不太了解，那么可以学习一下相关流的知识，注意在使用结束后不要忘记关闭流。

（3）下载方法。本文中例子只是对固定位置的文件进行下载，在实际应用中，可以根据具体情况进行修改。同样，下载方法也是使用流的方式，并且响应到浏览器。

FileController 类代码如代码清单 3-42 所示。

代码清单 3-42　上传和下载项目-FileController 文件内容

```
@RestController
public class FileController {

    private static final String filePath="/Users/dalaoyang/Downloads/";
    private static final Logger log = LoggerFactory.getLogger
(FileController.class);

    @RequestMapping(value = "/upload")
    public String upload(@RequestParam("file") MultipartFile file) {
        try {
            if (file.isEmpty()) {
                return "文件为空";
            }
            // 获取文件名
            String fileName = file.getOriginalFilename();
```

```
            log.info("上传的文件名为: " + fileName);
            // 设置文件存储路径
            String path = filePath + fileName;
            File dest = new File(path);
            // 检测是否存在目录
            if (!dest.getParentFile().exists()) {
                dest.getParentFile().mkdirs();// 新建文件夹
            }
            file.transferTo(dest);// 文件写入
            return "上传成功";
        } catch (IllegalStateException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return "上传失败";
    }

    @PostMapping("/batch")
    public String handleFileUpload(HttpServletRequest request) {
        List<MultipartFile> files = ((MultipartHttpServletRequest) request).
getFiles("file");
        MultipartFile file = null;
        BufferedOutputStream stream = null;
        for (int i = 0; i < files.size(); ++i) {
            file = files.get(i);
            if (!file.isEmpty()) {
                try {
                    byte[] bytes = file.getBytes();
                    stream = new BufferedOutputStream(new FileOutputStream(
                            new File(filePath + file.getOriginalFilename())));
                                                    //设置文件路径及名字
                    stream.write(bytes);            // 写入
                    stream.close();
                } catch (Exception e) {
                    stream = null;
                    return "第 " + i + " 个文件上传失败 ==> "
                            + e.getMessage();
                }
            } else {
                return "第 " + i
                        + " 个文件上传失败，因为文件为空";
            }
```

```
        }
        return "上传成功";
    }

    @GetMapping("/download")
    public String downloadFile( HttpServletResponse response) {
        String fileName = "dalaoyang.jpg";          // 文件名
        if (fileName != null) {
            //设置文件路径
            File file = new File(filePath+fileName);
            if (file.exists()) {
                response.setContentType("application/force-download");
                                              // 设置强制下载不打开
                response.addHeader("Content-Disposition", "attachment;
fileName=" + fileName);                       // 设置文件名
                byte[] buffer = new byte[1024];
                FileInputStream fis = null;
                BufferedInputStream bis = null;
                try {
                    fis = new FileInputStream(file);
                    bis = new BufferedInputStream(fis);
                    OutputStream os = response.getOutputStream();
                    int i = bis.read(buffer);
                    while (i != -1) {
                        os.write(buffer, 0, i);
                        i = bis.read(buffer);
                    }
                    return "下载成功";
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    if (bis != null) {
                        try {
                            bis.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                    if (fis != null) {
                        try {
                            fis.close();
                        } catch (IOException e) {
```

```
                                    e.printStackTrace();
                                }
                            }
                        }
                    }
                }
                return "下载失败";
            }
        }
```

本节只是进行简单的上传和下载，当然上述方法并不适用于大文件，只是对使用 FreeMarker 模板进行一个回顾。

3.9 小 结

本章从 Spring Boot 使用传统 Spring MVC 模式到 Spring 5 以后的 WebFlux 开始介绍，紧接着介绍配置文件、热部署等实用的内容，最后介绍模板框架，让读者可以在本章由浅入深地学习 Spring Boot 关于 Web 方面的使用，对 Spring Boot 关于 Web 方面的内容有深刻的认识，并能够运用自如。

第 4 章

Spring Boot 的数据库之旅

数据库是存储管理数据的仓库，是开发一个应用的必要因素。其实从某种程度上来说，数据库是实现一个系统的根本，甚至有时我们可以理解为：应用实质上就是展示数据库、存储数据库数据等一系列对数据库的操作，所以学习数据库操作对我们来说尤其重要。本章将学习 Spring Boot 对数据库的操作，让我们开启 Spring Boot 的数据库之旅。

4.1 使用数据库

数据库分为两种，即关系型数据库和非关系型数据库。关系型数据库是指通过关系模型组织数据的数据库，并且可以利用外键等保持一致性；而非关系型数据库其实不像是数据库，更像是一种以 key-value 模式存储对象的结构。本节来了解 Spring Boot 如何使用数据库，以依赖和配置文件为例，后续章节会对数据库进行具体使用。

4.1.1 使用 MySQL 数据库

MySQL 数据库（官网地址：https://www.mysql.com）是一种关系型数据库，由瑞典的一家公司开发，现在是 Oracle 公司旗下的产品。MySQL 使用 C 和 C++语言开发，提供多种存储引擎，提供多种连接途径，例如 ODBC、JDBC、TCP/IP 等，并且支持多线程，是当今最流行的数据库之一，并且免费提供给开发者使用。MySQL 数据库的 LOGO 使用一个海豚作为标记，如图 4-1 所示。海豚标志的名字叫 sakila，它是由 MySQL 的创始人从用户在“海豚命名”的竞赛中建议的大量名字表中选出的。



图 4-1　MySQL 数据库 LOGO

同时，MySQL 数据库是一个高性能的数据库，并且支持多种开发语言，如 C、C++、Python、Java、Perl、PHP、Eiffel、Ruby 和 Tcl 等。并且，MySQL 支持大型的数据库，可以处理拥有上千万条记录的大型数据库。MySQL 提供多种存储引擎及索引格式，采用 GPL 协议，如果有需要的话，可以根据场景修改源码来开发自己的 MySQL 系统。

在 Spring Boot 中使用 MySQL 很简单，大致分为两步：

（1）在 pom 文件中加入依赖，如代码清单 4-1 所示。

代码清单 4-1 MySQL 依赖代码

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

（2）在配置文件中配置数据库信息，如代码清单 4-2 所示。

代码清单 4-2 Mysql 配置文件代码

```
##数据库地址
spring.datasource.url=jdbc:mysql://localhost:3306/test?characterEncoding=
utf8&useSSL=false
##数据库用户名
spring.datasource.username=root
##数据库密码
spring.datasource.password=root
##数据库驱动
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

4.1.2 使用 SQL Server 数据库

SQL Server 是微软公司推出的关系型数据库（官网地址：https://www.microsoft.com/zh-cn/sql-server），最初是由 Microsoft、Sybase 和 Ashton-Tate 三家公司共同开发的，于 1988 年推出了第一个 OS/2 版本。在 Windows NT 推出后，Microsoft 与 Sybase 在 SQL Server 的开发上就分道扬镳了。Microsoft 将 SQL Server 移植到 Windows NT 系统上，专注于开发推广 SQL Server 的 Windows NT 版本；Sybase 则较专注于 SQL Server 在 UNIX 操作系统上的应用。SQL Server 与 MySQL 有很多相似的地方，可以跨越多种平台使用，并且提供了更安全可靠的存储功能，方便构建高可用性能的应用程序。SQL Server 数据库的 LOGO 使用微软公司一贯的风格，如图 4-2 所示。



图 4-2 SQL Server 数据库的 LOGO

SQL Server 数据库同样提供了很多优点，如易用性、适合分布式组织的可伸缩性、用于决策支持的数据仓库功能、与许多其他服务器软件紧密关联的集成性、良好的性价比等。但是相较于MySQL，其有一定的缺点，如局限性（只能运行在 Windows 系统上）、当连接数过高时性能不够稳定等。

Spring Boot 使用 SQL Server 数据库也很简单，分为两步：

（1）在 pom 文件中加入依赖，如代码清单 4-3 所示。

代码清单 4-3　SQL Server 依赖代码

```
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <scope>runtime</scope>
</dependency>
```

（2）在配置文件中配置数据库信息，如代码清单 4-4 所示。

代码清单 4-4　SQL Server 配置文件代码清单

```
##数据库地址
spring.datasource.url=jdbc:sqlserver://192.168.16.218:1433;databaseName=d
ev_btrpawn
##数据库用户名
spring.datasource.username=sa
##数据库密码
spring.datasource.password=p@ssw0rd
##数据库驱动
spring.datasource.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServe
rDriver
```

4.1.3　使用 Oracle 数据库

Oracle Database（官网地址：https://www.oracle.com）又名Oracle RDBMS，简称 Oracle。Oracle 是甲骨文公司的一款关系数据库管理系统，它在数据库领域一直处于领先地位。Oracle 数据库系统是目前世界上流行的关系数据库管理系统，系统可移植性好、使用方便、功能性强，适用于各类大、中、小、微机环境。它是一种高效率的、可靠性好的、适应高吞吐量的数据库解决方案。Oracle 数据库的 LOGO 很简单，就是 Oracle 公司的图标，如图 4-3 所示。



图 4-3　Oracle 数据库的 LOGO

Spring Boot 使用 Oracle 数据库需要自行下载依赖 JAR 包，中央仓库没有对应的 JAR 包，引入 JAR 之后在配置文件中加入如下配置，如代码清单 4-5 所示。

代码清单 4-5　Oracle 配置文件

```
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=dalaoyang
spring.datasource.password=dalaoyang123
```

这里延伸一个关于 Maven 的小技巧，主要介绍常用的导入 Jar 的方式：

1. 命令行方式

命令行的方式比较简单，比如我们要导入在本地下载文件夹中的 ojdbc8.jar 到本地仓库，如代码清单 4-6 所示。

代码清单 4-6　命令行导入 JAR

```
  mvn install:install-file -Dfile=/Users/dalaoyang/Downloads/ojdbc8.jar
-DgroupId=com.oracle -DartifactId=oracle -Dversion=8.0.0 -Dpackaging=jar
```

其中：

- -Dfile：文件位置。
- -DgroupId：依赖的 groupId。
- -DartifactId：依赖的 artifactId。
- -Dversion：依赖的版本号。
- -Dpackaging：什么类型的文件（这里使用 Jar）。

执行命令后如图 4-4 所示。

```
~/Downloads > mvn install:install-file -Dfile=/Users/dalaoyang/Downloads/ojdbc8.jar -DgroupId=com.oracle -DartifactId=orac
le -Dversion=8.0.0 -Dpackaging=jar
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-install-plugin:2.4:install-file (default-cli) @ standalone-pom ---
[INFO] Installing /Users/dalaoyang/Downloads/ojdbc8.jar to /Users/dalaoyang/maven_repository/com/oracle/oracle/8.0.0/oracle
-8.0.0.jar
[INFO] Installing /var/folders/7g/smtn3btx1v598j38dfqd3z7h0000gp/T/mvninstall6053533590357082494.pom to /Users/dalaoyang/ma
ven_repository/com/oracle/oracle/8.0.0/oracle-8.0.0.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.556 s
[INFO] Finished at: 2019-04-25T15:36:15+08:00
[INFO] Final Memory: 6M/155M
[INFO] ------------------------------------------------------------------------
```

图 4-4　命令行导入 Jar 执行图

然后我们查看一下本地 Maven 仓库，也可以找到刚刚上传的 Jar，如图 4-5 所示。

```
~/maven_repository/com/oracle/oracle/8.0.0 ll
total 8152
-rw-r--r--  1 dalaoyang  staff   178B  4 25 15:36 _remote.repositories
-rw-r--r--  1 dalaoyang  staff   4.0M  4 25 15:19 oracle-8.0.0.jar
-rw-r--r--  1 dalaoyang  staff   460B  4 25 15:36 oracle-8.0.0.pom
~/maven_repository/com/oracle/oracle/8.0.0
```

图 4-5　查看本地 Maven 仓库

导入后我们就可以在本地正常引用刚刚导入的 Jar 文件了。

2. 引用本地 Jar

命令行的方式看起来似乎有一些麻烦，不过不要紧，Spring Boot 提供了引用本地 Jar 文件的方式，比如在本地项目 src/lib 目录下有一个 ojdbc8.jar，我们只需要在 pom.xml 文件中配置如下内容，如代码清单 4-7 所示。

代码清单 4-7　POM 引入本地 Jar

```
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>oracle</artifactId>
    <version>8.0.0</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/src/lib/ojdbc8.jar</systemPath>
</dependency>
```

配置后重新导入 Jar，如图 4-6 所示。

```
▼ Dependencies
  ▶ org.springframework.boot:spring-boot-starter:2.1.4.RELEASE
  ▶ org.springframework.boot:spring-boot-starter-test:2.1.4.RELEASE (test)
    com.oracle:oracle:8.0.0 (system)
```

图 4-6　查看本地 Maven 仓库

可以看到，这种方式在本地也可以引用本地 Jar 文件。不过有一个问题，这种方式在使用 Spring Boot 项目打 Jar 包的时候并没有将我们的本地 Jar 导入，那么怎么解决呢？

其实 Spring Boot 提供了插件解决这个问题，我们只需要在 pom.xml 文件中引入如下插件即可解决，如代码清单 4-8 所示。

代码清单 4-8　POM 引入 Spring Boot 打包本地 Jar 插件

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <includeSystemScope>true</includeSystemScope>
    </configuration>
</plugin>
```

3. 使用 Nexus 平台导入

这里所说的 Nexus 平台就是本地私服，与 Maven 中央仓一致。访问私服地址并登录，如图 4-7 所示。

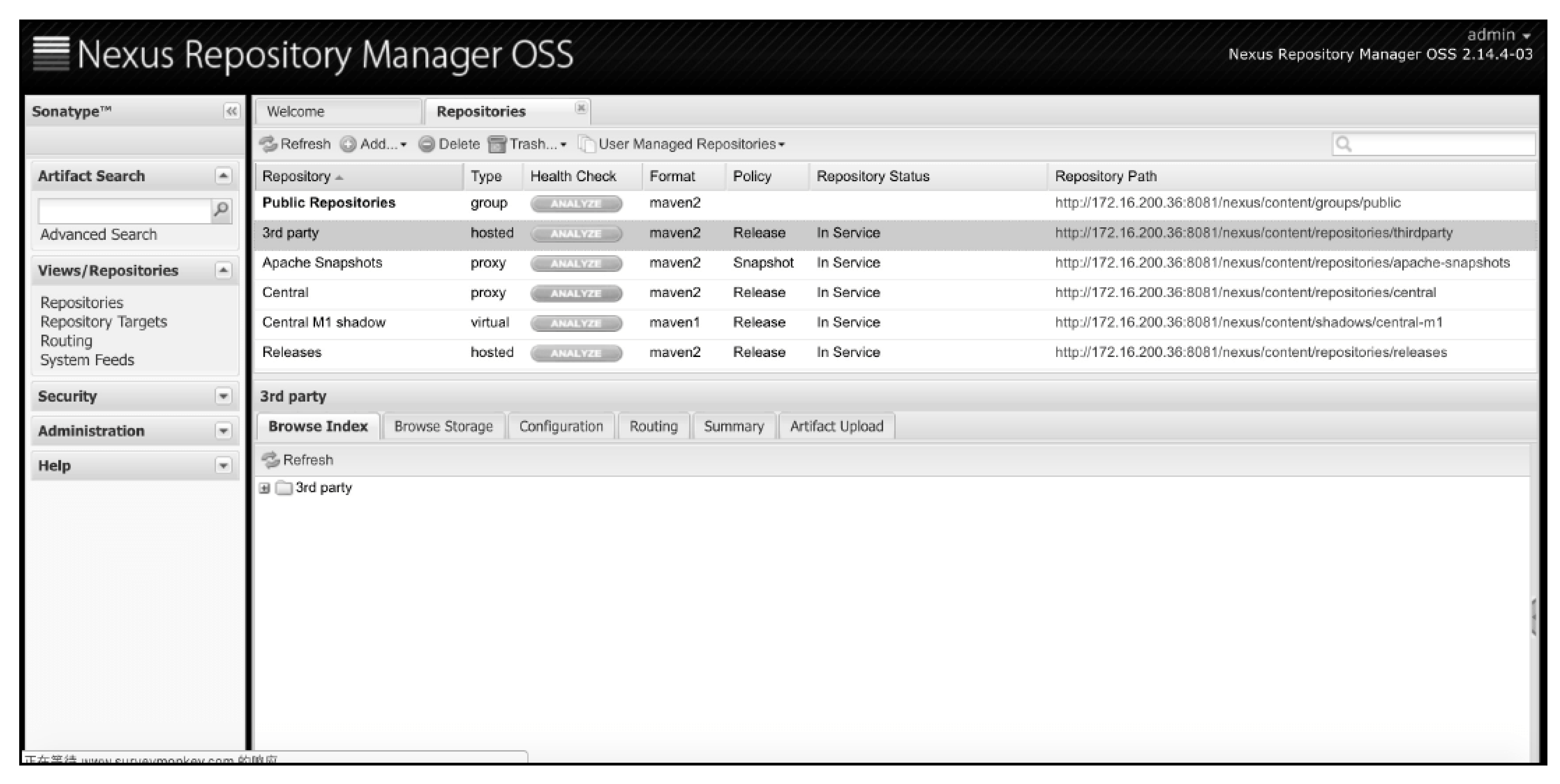


图 4-7 查看本地 Nexus 平台

这里以上传第三方 Jar 为例，单击 Artifact Upload 按钮，如图 4-8 所示。

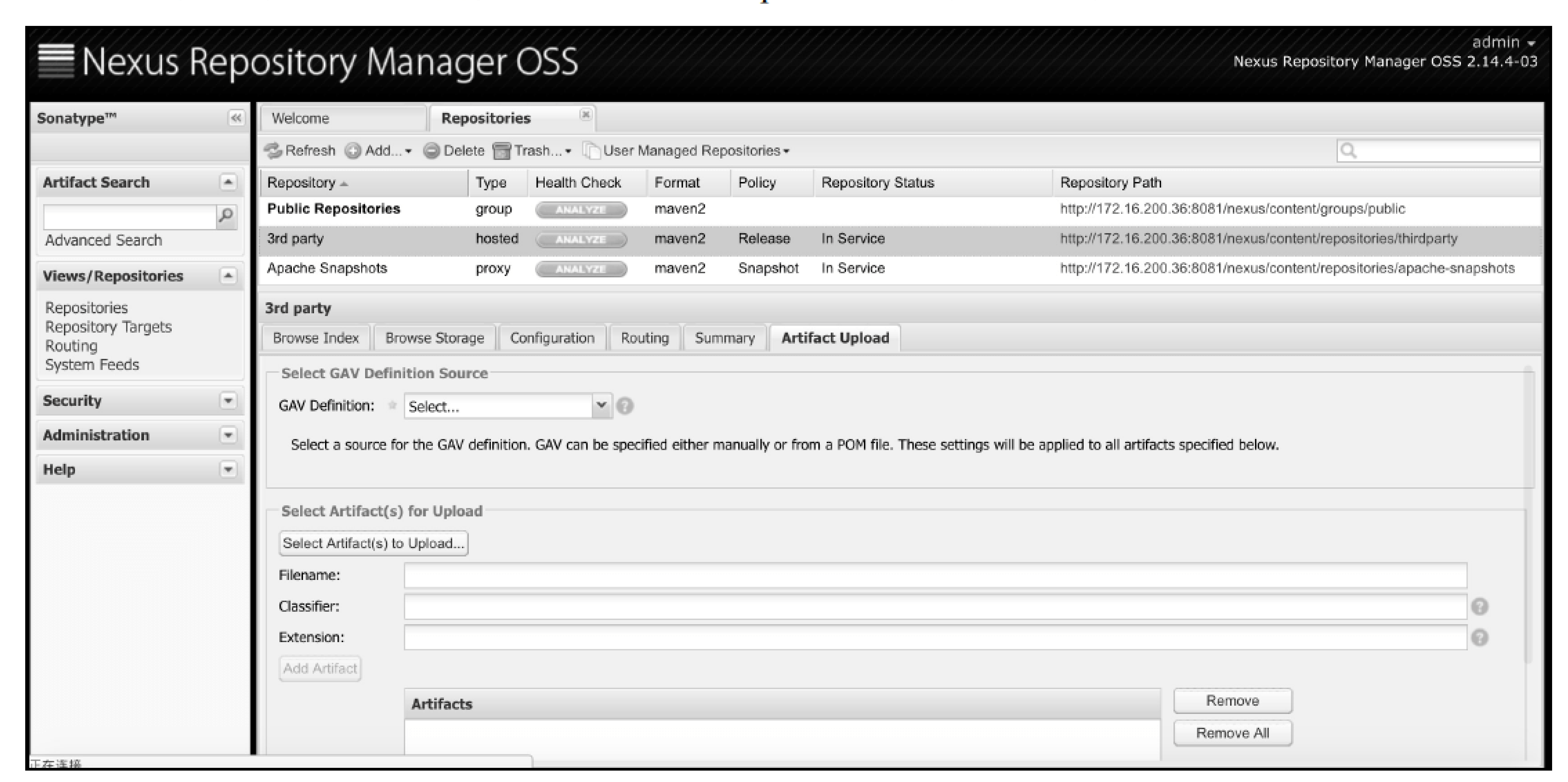


图 4-8 查看本地 Maven 仓库-Artifact Upload

在 GAV Definition 下拉框中选择 GAV Parameters，如图 4-9 所示。

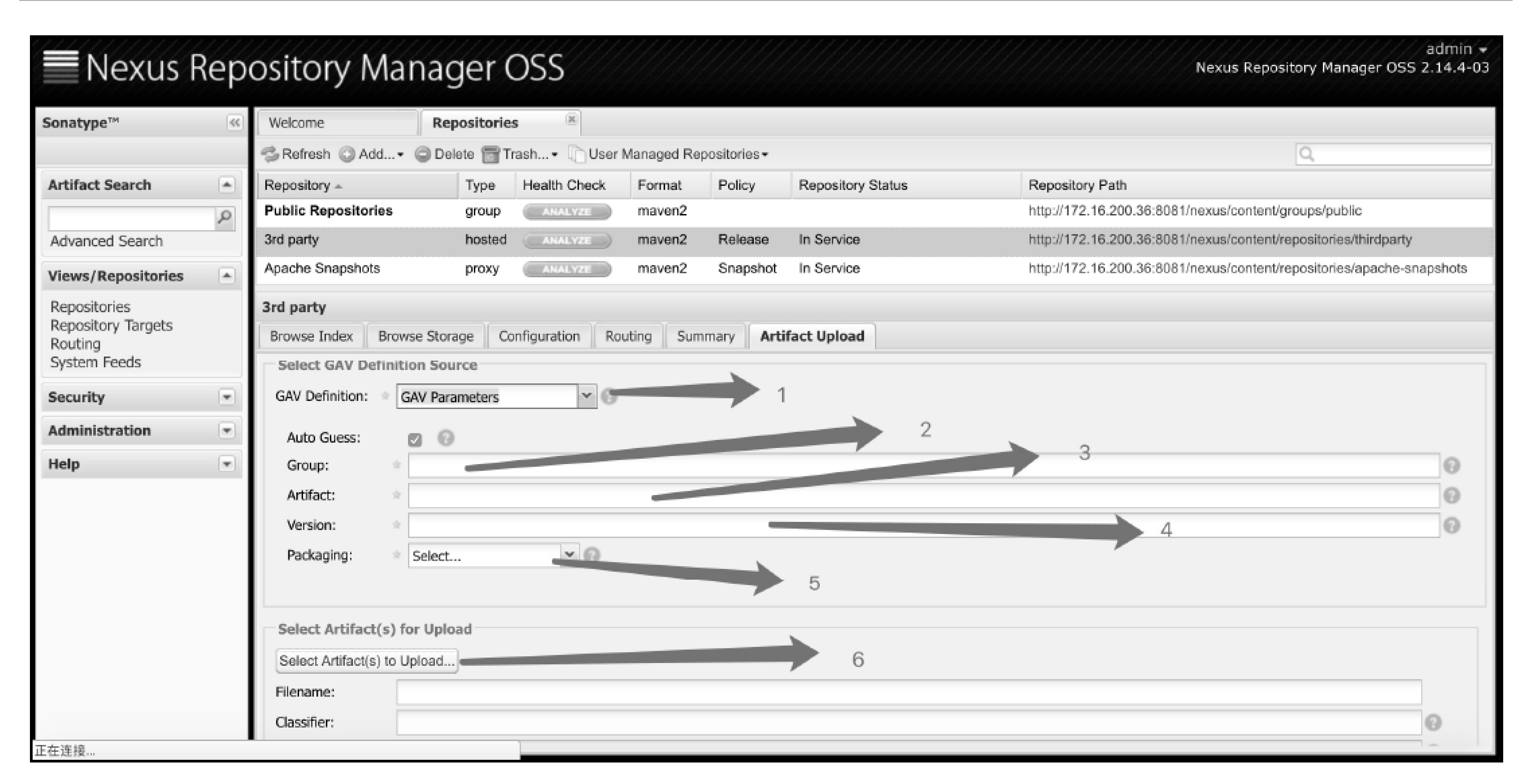


图 4-9　查看本地 Maven 仓库-GAV Parameters

分别填入 Group、Artifact、Version 并且选择合适的 Packaging，设置完成后单击 Select Artifact(s) for Upload 按钮，选择 Jar 文件，都完成后单击左下方的 Add Artifact 按钮，最后单击 Upload Artifact(s) 完成上传，结果与使用命令行一致。

4.1.4　使用 MongoDB 数据库

MongoDB（官网地址：https://www.mongodb.com）是一种非关系型数据库，它是一个基于分布式文件存储的数据库。MongoDB 由 C++语言编写，有高性能、容易部署等优点，官网的介绍如下：

MongoDB（来自于英文单词 Humongous，中文含义为“庞大”）是可以应用于各种规模的企业、各个行业以及各类应用程序的开源数据库。作为一个适用于敏捷开发的数据库，MongoDB 的数据模式可以随着应用程序的发展而灵活地更新。与此同时，它为开发人员提供了传统数据库的功能：二级索引、完整的查询系统以及严格一致性等。MongoDB 能够使企业更加具有敏捷性和可扩展性，各种规模的企业都可以通过使用 MongoDB 来创建新的应用，提高与客户之间的工作效率、加快产品上市时间以及降低企业成本。

MongoDB 是专为可扩展性、高性能和高可用性而设计的数据库。它可以从单服务器部署扩展到大型、复杂的多数据中心架构。利用内存计算的优势，MongoDB 能够提供高性能的数据读写操作。MongoDB 的本地复制和自动故障转移功能使你的应用程序具有企业级的可靠性和操作灵活性。

MongoDB 数据库的 LOGO 使用一个绿色叶子和 mongoDB 英文字母组成，如图 4-10 所示。



图 4-10　MongoDB 数据库 LOGO

Spring Boot 使用 MongoDB 数据库很简单，分为两步：

（1）在 pom 文件加入依赖，如代码清单 4-9 所示。

代码清单 4-9　MongoDB 依赖代码

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
    </dependency>
</dependencies>
```

（2）在配置文件中加入 MongoDB 配置，如代码清单 4-10 所示。

代码清单 4-10　MongoDB 配置文件

```
##无密码
spring.data.mongodb.uri=mongodb://localhost:27017/test
##有密码
#spring.data.mongodb.uri=mongodb://root(userName):root(password)@localhos
t(ip 地址):27017(端口号)/test(collections/数据库)
```

4.1.5　使用 Neo4j 数据库

Neo4j（官网地址：https://neo4j.com/）是一种非关系型数据库，它是一种图形数据库，将结构化数据存储在网络上而不是表中，同时可以享受到事务特性等优势。Neo4j 数据库的 LOGO 如图 4-11 所示。



图 4-11　Neo4j 数据库的 LOGO

Spring Boot 使用 Neo4j 数据库很简单，分为两步：

（1）在 pom 文件中加入依赖，如代码清单 4-11 所示。

代码清单 4-11　Neo4j 依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-neo4j</artifactId>
</dependency>
```

（2）在配置文件中加入配置，如代码清单 4-12 所示。

代码清单 4-12　Neo4j 配置文件代码

```
spring.data.neo4j.uri=http://localhost:7474
spring.data.neo4j.username=dalaoyang
spring.data.neo4j.password=dalaoyang123
```

4.1.6　使用 Redis 数据库

Redis（官网地址：https://redis.io/）是一种非关系型数据库，使用 ANSI C 语言开发，是一种 Key-Value 模式的数据库，支持多种 value 类型，如 string（字符串）、list（链表）、set（集合）、zset（sorted set，有序集合）和 hash（哈希类型）。如图 4-12 所示是 Redis 数据库的 LOGO。



图 4-12　Redis 数据库的 LOGO

对于 Redis 来说，我们可能对它使用的更多的是缓存，毕竟它可以高效地对数据进行操作。其实它还具备很多功能，比如消息队列、发布、订阅消息等。另外，它提供了持久化的方式，在后续章节有详细介绍，这里先简单对依赖和配置进行介绍。Spring Boot 使用 Redis 数据库分为两步：

（1）在 pom 文件中加入依赖，如代码清单 4-13 所示。

代码清单 4-13　Redis 依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

（2）在配置文件中加入配置，如代码清单 4-14 所示。

代码清单 4-14　Redis 配置文件代码

```
# Redis 数据库索引（默认为 0）
spring.redis.database=0
# Redis 服务器地址
spring.redis.host=localhost
# Redis 服务器连接端口
spring.redis.port=6379
# Redis 服务器连接密码（默认为空）
spring.redis.password=
#连接池最大连接数（使用负值表示没有限制）
spring.redis.pool.max-active=8
# 连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.pool.max-wait=-1
# 连接池中的最大空闲连接
spring.redis.pool.max-idle=8
# 连接池中的最小空闲连接
spring.redis.pool.min-idle=0
# 连接超时时间（毫秒）
spring.redis.timeout=0
```

4.1.7 使用 Memcached 数据库

Memcached（官网地址：http://memcached.org/）是一个高性能的分布式内存对象缓存系统，用于动态 Web 应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。Memcached 基于一个存储键/值对的 hashmap。其守护进程（daemon）是用 C 写的，但是客户端可以用任何语言来编写，并通过 Memcached 协议与守护进程通信。Memcached 数据库的 LOGO 如图 4-13 所示。



图 4-13　Memcached 数据库的 LOGO

在后续有专门的章节介绍 Memcached，这里和 Redis 一样，只是对依赖和配置进行介绍。Spring Boot 使用 Memcached 数据库分为两步：



（1）在 pom 文件中加入依赖，如代码清单 4-15 所示。

代码清单 4-15　Memcached 依赖代码

```
<dependency>
    <groupId>net.spy</groupId>
    <artifactId>spymemcached</artifactId>
    <version>2.12.2</version>
</dependency>
```

（2）在配置文件中加入配置，如代码清单 4-16 所示。

代码清单 4-16　Memcached 配置文件代码

```
# memcached 地址
memcache.ip=localhost
# memecached 端口
memcache.port=11211
```

本节介绍了 Spring Boot 对关系型数据库及非关系型数据库的使用，只是介绍了关于配置和依赖，在具体使用上其实大致都相同。接下来以 MySQL 为例，具体介绍如何操作数据库。

4.2　使用 JDBC 操作数据库

JDBC 的全名是 Java DataBase Connectivity，可能是我们最先接触到的数据库连接，通过 JDBC 可以直接使用 Java 编程来操作数据库。其实我们可以这样理解 JDBC，它就是一个可以执行 SQL 语句的 Java API。

4.1 节讲了很多数据库，本节以 MySQL 为例介绍 Spring Boot 使用 JDBC 操作 MySQL 数据库。

4.2.1　JDBC 依赖配置

新建项目，在 pom 文件中加入 JDBC 依赖和 MySQL 依赖以及 Web 功能的依赖，如代码清单 4-17 所示。

代码清单 4-17　JDBC 项目依赖代码

```
<!-- WEB 依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- jdbc-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<!-- mysql-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

4.2.2　配置数据库信息

在配置文件中配置数据库信息，与 4.1 节介绍的一样，配置文件内容如代码清单 4-18 所示。

代码清单 4-18　JDBC 项目配置代码

```
##数据库地址
spring.datasource.url=jdbc:mysql://localhost:3306/test?characterEncoding=
utf8&useSSL=false
##数据库用户名
spring.datasource.username=root
##数据库密码
spring.datasource.password=root
##数据库驱动
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

4.2.3 创建实体类

创建一个实体类，用作接收表数据，这里省略 set、get 方法，代码如代码清单 4-19 所示。

代码清单 4-19 JDBC 项目 User 实体类代码

```
public class User {
    private int id;
    private String user_name;
    private String user_password;
    public User(int id, String user_name, String user_password) {
        this.id = id;
        this.user_name = user_name;
        this.user_password = user_password;
    }
    public User() {
    }

    ...
```

4.2.4 使用 Controller 进行测试

新建一个 UserController，由于这个类只是用来测试使用 JDBC 操作数据库，因此我们在类上加入@RestController 注解，熟悉 Spring 的人都知道，这个注解其实相当于@ResponseBody 和@Controller 两个注解结合起来使用。并且在 UserController 内注入 JdbcTemplate，代码如代码清单 4-20 所示。

代码清单 4-20 JDBC 项目配置代码

```
@RestController
public class UserController {
   @Autowired
   private JdbcTemplate jdbcTemplate;

   ...
```

其中，@Autowired 注解是 Spring 用于自动装配类的注解，功能和@Resource 类似。刚刚我们注入的 JdbcTemplate 就是 Spring Boot 使用 JDBC 操作数据库的核心，接下来进一步对它进行学习。

对于操作数据库来说，其实基本上就是创建（Create）、更新（Update）、读取（Retrieve）和删除（Delete）操作。而对于 JdbcTemplate 操作数据库的 CURD，基本上分为三种方法：

1. execute 方法

execute 方法用来直接执行 SQL 语句，是最直接的操作数据库的方法。接下来我们在 UserController 内写一个建表方法 createTable 来使用它，方法内容如代码清单 4-21 所示。

代码清单 4-21 JDBC 项目 execute 方法代码

```
@GetMapping("createTable")
public String createTable(){
    String sql =
    "CREATE TABLE `user` (\n" +
    "  `id` int(11) NOT NULL AUTO_INCREMENT,\n" +
    "  `user_name` varchar(255) NOT NULL,\n" +
    "  `user_password` varchar(255) DEFAULT NULL,\n" +
    "  PRIMARY KEY (`id`)\n" +
    ") ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=latin1;";
    jdbcTemplate.execute(sql);
    return "创建 User 表成功";
}
```

启动项目，在浏览器上访问 http://localhost:8080/createTable，显示如下。然后查看数据库，顺利地创建了 User 表。这里是以创建表为例，其实也可以直接执行 CURD 操作，就不一一举例了。

```
创建 User 表成功
```

2. update 方法

update 方法多用于增、删、改操作，update 方法默认返回一个 int 值，了解 SQL 的人应该知道，方法的返回值就是影响的数据行数，比如我们数据库中存在两条性别为男的用户数据，当执行 update 语句修改性别为男的数据时，执行成功后，我们可以得到返回值 2，这个就是我们执行 update 方法的返回值，新增、删除操作的原理类似。当然，也支持直接执行 SQL 语句，比如下面的方法 saveUserSql，直接执行一个插入语句，代码如代码清单 4-22 所示。

代码清单 4-22 JDBC 项目 update 方法代码

```
@GetMapping("saveUserSql")
public String saveUserSql(){
    String sql = "INSERT INTO USER (USER_NAME,USER_PASSWORD) VALUES
('dalaoyang','123')";
    int rows= jdbcTemplate.update(sql);
    return "执行成功，影响"+rows+"行";
}
```

重启项目，在浏览器访问可以看到：

```
执行成功，影响 1 行
```

上面的场景可能并不常用，甚至基本上用不到，因为插入的数据不可能是已知的并且都是固定的。插入的数据是动态的怎么办呢？update 方法中也是支持传值进去的，只需要在执行的 SQL 上用问号来代替参数，其中 update 方法内第一个参数是执行的 SQL，接着对应传入动态的参数即可。下面 3 个方法分别列举了增、删、改的方法，代码如代码清单 4-23 所示。

代码清单 4-23　JDBC 项目增加、删除、修改方法代码

```
    //新增方法
    @GetMapping("saveUser")
    public String saveUser(String userName,String passWord){
        int rows= jdbcTemplate.update("INSERT INTO USER (USER_NAME,USER_PASSWORD)
VALUES (?,?)",userName,passWord);
        return "执行成功，影响"+rows+"行";
    }

    //修改方法
    @GetMapping("updateUserPassword")
    public String updateUserPassword(int id,String passWord){
        int rows= jdbcTemplate.update("UPDATE USER SET USER_PASSWORD = ? WHERE ID
= ?",passWord,id);
        return "执行成功，影响"+rows+"行";
    }

    //删除方法
    @GetMapping("deleteUserById")
    public String deleteUserById(int id){
        int rows= jdbcTemplate.update("DELETE FROM  USER  WHERE ID = ?",id);
        return "执行成功，影响"+rows+"行";
    }
```

这里就不一一测试了，感兴趣的读者可以自行测试。接下来我们继续学习 update 方法的延伸，其实 JdbcTemplate 中也提供了批处理方法 batchUpdate，可以传入 SQL 和一个批处理的数组进行操作。比如下面的 batchSaveUserSql 方法，可以批量插入 10 条数据，代码如代码清单 4-24 所示。

代码清单 4-24　JDBC 项目 batchSaveSql 方法代码

```
    @GetMapping("batchSaveUserSql")
    public String batchSaveUserSql(){
        String sql =
        "INSERT INTO USER (USER_NAME,USER_PASSWORD) VALUES (?,?)" ;
        List<Object[]> paramList = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            String[] arr = new String[2];
            arr[0] = "zhangsan"+i;
            arr[1] = "password"+i;
```

```
            paramList.add(arr);
        }
        jdbcTemplate.batchUpdate(sql,paramList);
        return "执行成功";
    }
```

3. query 方法

query 方法看名字就能想到，用于执行有关查询的方法。

首先我们使用一个 JdbcTemplate 的 query 方法，这里可以用到之前创建的实体类 User，方法如下，根据 userName 查询一个列表，代码如代码清单 4-25 所示。

代码清单 4-25　JDBC 项目 query 方法代码

```
    @GetMapping("getUserByUserName")
    public List getUserByUserName(String userName){
        String sql = "SELECT * FROM USER WHERE USER_NAME = ?";
        List<User> list= jdbcTemplate.query(sql,new Object[]{userName},new
BeanPropertyRowMapper<>(User.class));
        return list;
    }
```

重启项目，在浏览器上访问 http://localhost:8080/getUserByUserName?userName=zhangsan0，如下所示：

```
[{"id":10,"user_name":"zhangsan0","user_password":"password0"}]
```

刚刚我们使用了一个返回 List 集合的方法，接下来使用一个返回 Map 的方法 queryForMap，代码如代码清单 4-26 所示。

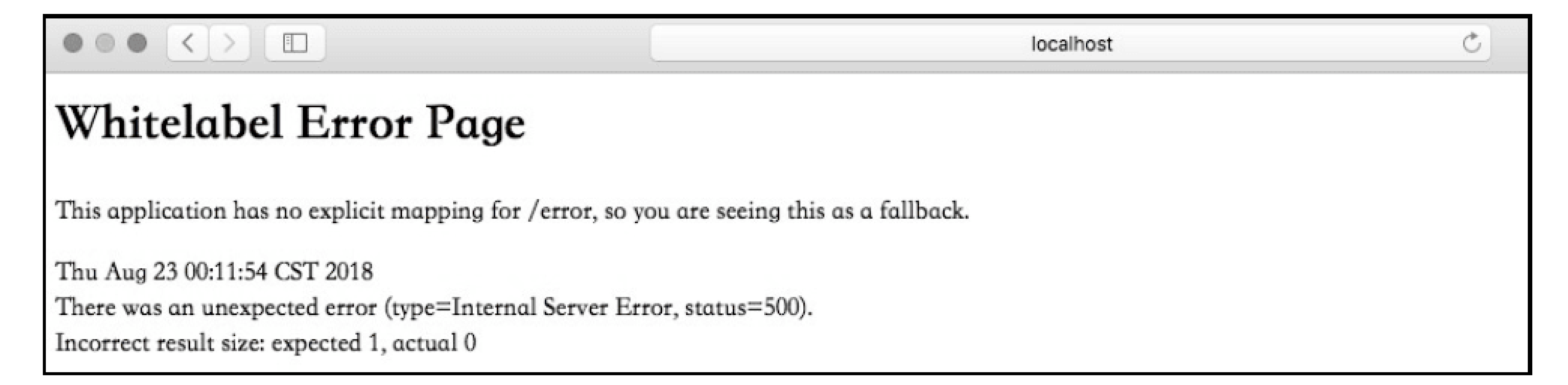
代码清单 4-26　JDBC 项目 execute 方法代码

```
    @GetMapping("getMapById")
    public Map getMapById(Integer id){
        String sql = "SELECT * FROM USER WHERE ID = ?";
        Map map= jdbcTemplate.queryForMap(sql,id);
        return map;
    }
```

在浏览器访问 http://localhost:8080/getMapById?id=1，如下所示：

```
{"id":1,"user_name":"lisi","user_password":"111"}
```

接下来我们查询一个数据库中不存在的数据，在浏览器访问 http://localhost:8080/getMapById?id=1000，页面显示如图 4-14 所示。



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Thu Aug 23 00:11:54 CST 2018
There was an unexpected error (type=Internal Server Error, status=500).
Incorrect result size: expected 1, actual 0

图 4-14 JDBC 项目 Error 图

我们回过头看一下控制台，报错如代码清单 4-27 所示。

代码清单 4-27 JDBC 项目控制台错误代码

```
   org.springframework.dao.EmptyResultDataAccessException: Incorrect result
size: expected 1, actual 0
      at org.springframework.dao.support.DataAccessUtils.
nullableSingleResult(DataAccessUtils.java:97)
~[spring-tx-5.0.7.RELEASE.jar:5.0.7.RELEASE]
      at org.springframework.jdbc.core.JdbcTemplate.queryForObject
(JdbcTemplate.java:772) ~[spring-jdbc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
      at org.springframework.jdbc.core.JdbcTemplate.queryForMap
(JdbcTemplate.java:807) ~[spring-jdbc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
      at com.dalaoyang.controller.UserController.getMapById
(UserController.java:104) ~[classes/:na]
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
~[na:1.8.0_131]
      at sun.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:62) ~[na:1.8.0_131]
      at sun.reflect.DelegatingMethodAccessorImpl.invoke
(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0_131]
      at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0_131]
      at org.springframework.web.method.support.InvocableHandlerMethod.
doInvoke(InvocableHandlerMethod.java:209)
~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
      at org.springframework.web.method.support.InvocableHandlerMethod.
invokeForRequest(InvocableHandlerMethod.java:136)
~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
      at
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandle
rMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:102)
~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
      at org.springframework.web.servlet.mvc.method.annotation.
RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapte
r.java:877) ~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
```

```
	at org.springframework.web.servlet.mvc.method.annotation.
RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.jav
a:783) ~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at org.springframework.web.servlet.mvc.method.
AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87)
~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at org.springframework.web.servlet.DispatcherServlet.doDispatch
(DispatcherServlet.java:991) ~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at org.springframework.web.servlet.DispatcherServlet.doService
(DispatcherServlet.java:925) ~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at org.springframework.web.servlet.FrameworkServlet.processRequest
(FrameworkServlet.java:974) ~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at org.springframework.web.servlet.FrameworkServlet.doGet
(FrameworkServlet.java:866) ~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at javax.servlet.http.HttpServlet.service(HttpServlet.java:635)
~[tomcat-embed-core-8.5.31.jar:8.5.31]
	at org.springframework.web.servlet.FrameworkServlet.service
(FrameworkServlet.java:851) ~[spring-webmvc-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at javax.servlet.http.HttpServlet.service(HttpServlet.java:742)
~[tomcat-embed-core-8.5.31.jar:8.5.31]
	at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
(ApplicationFilterChain.java:231) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
	at org.apache.catalina.core.ApplicationFilterChain.doFilter
(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
	at org.apache.tomcat.websocket.server.WsFilter.doFilter
(WsFilter.java:52) ~[tomcat-embed-websocket-8.5.31.jar:8.5.31]
	at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
	at org.apache.catalina.core.ApplicationFilterChain.doFilter
(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
	at org.springframework.web.filter.RequestContextFilter.
doFilterInternal(RequestContextFilter.java:99)
~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at org.springframework.web.filter.OncePerRequestFilter.doFilter
(OncePerRequestFilter.java:107) ~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
	at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
	at org.apache.catalina.core.ApplicationFilterChain.doFilter
(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
	at org.springframework.web.filter.HttpPutFormContentFilter.
doFilterInternal(HttpPutFormContentFilter.java:109)
~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
```

```
        at org.springframework.web.filter.OncePerRequestFilter.doFilter
(OncePerRequestFilter.java:107) ~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.core.ApplicationFilterChain.doFilter
(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.springframework.web.filter.HiddenHttpMethodFilter.
doFilterInternal(HiddenHttpMethodFilter.java:93)
~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
        at org.springframework.web.filter.OncePerRequestFilter.doFilter
(OncePerRequestFilter.java:107) ~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.core.ApplicationFilterChain.doFilter
(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.springframework.web.filter.CharacterEncodingFilter.
doFilterInternal(CharacterEncodingFilter.java:200)
~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
        at org.springframework.web.filter.OncePerRequestFilter.doFilter
(OncePerRequestFilter.java:107) ~[spring-web-5.0.7.RELEASE.jar:5.0.7.RELEASE]
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
(ApplicationFilterChain.java:193) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.core.ApplicationFilterChain.doFilter
(ApplicationFilterChain.java:166) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.core.StandardWrapperValve.invoke
(StandardWrapperValve.java:198) ~[tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.core.StandardContextValve.invoke
(StandardContextValve.java:96) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke
(AuthenticatorBase.java:496) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.core.StandardHostValve.invoke
(StandardHostValve.java:140) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.valves.ErrorReportValve.invoke
(ErrorReportValve.java:81) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.core.StandardEngineValve.invoke
(StandardEngineValve.java:87) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.catalina.connector.CoyoteAdapter.service
(CoyoteAdapter.java:342) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.coyote.http11.Http11Processor.service
(Http11Processor.java:803) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.coyote.AbstractProcessorLight.process
(AbstractProcessorLight.java:66) [tomcat-embed-core-8.5.31.jar:8.5.31]
```

```
        at org.apache.coyote.AbstractProtocol$ConnectionHandler.process
(AbstractProtocol.java:790) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun
(NioEndpoint.java:1468) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at org.apache.tomcat.util.net.SocketProcessorBase.run
(SocketProcessorBase.java:49) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at java.util.concurrent.ThreadPoolExecutor.runWorker
(ThreadPoolExecutor.java:1142) [na:1.8.0_131]
        at java.util.concurrent.ThreadPoolExecutor$Worker.run
(ThreadPoolExecutor.java:617) [na:1.8.0_131]
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run
(TaskThread.java:61) [tomcat-embed-core-8.5.31.jar:8.5.31]
        at java.lang.Thread.run(Thread.java:748) [na:1.8.0_131]
```

接下来查看一下 DataAccessUtils 源代码，可以看到 nullableSingleResult 在查到空集合的时候默认抛出 EmptyResultDataAccessException 异常，看到就明白了。我们修改一下 getMapById 方法，在方法内捕获 EmptyResultDataAccessException 异常，方法代码修改如代码清单 4-28 所示。

代码清单 4-28　JDBC 项目 getMapById 方法代码

```
    @GetMapping("getMapById")
    public Map getMapById(Integer id){
        String sql = "SELECT * FROM USER WHERE ID = ?";
        Map map = null;
        try{
        map= jdbcTemplate.queryForMap(sql,id);
        }catch (EmptyResultDataAccessException e){
        return null;
        }
        return map;
    }
```

前面介绍了返回 List 集合、Map 集合，接下来学习一个返回 User 实体类的方法 queryForObject。基于刚刚查不到结果抛出异常的原因，吃一堑长一智，在这个方法中直接捕获这个异常，方法如代码清单 4-29 所示。

代码清单 4-29　JDBC 项目 getUserById 方法代码

```
    @GetMapping("getUserById")
    public User getUserById(Integer id){
        String sql = "SELECT * FROM USER WHERE ID = ?";
        User user= null;
        try{
        user = jdbcTemplate.queryForObject(sql,new Object[]{id},new
BeanPropertyRowMapper<>(User.class));
```

```
        }catch (EmptyResultDataAccessException e){
        return null;
        }
        return user;
    }
```

测试就不再赘述了，和之前的方法一样。到这里，Spring Boot 使用 JDBC 就告一段落了，毕竟在实际开发中对 JDBC 的使用不是很多，有一定基础就可以了。

4.3 使用 JPA 操作数据库

4.3.1 JPA 介绍

JPA 是 Java Persistence API 的简称，是 JCP 组织发布的 Java EE 标准之一。JPA 是一种面向对象的查询语言，定义了独特的 JPQL（Java Persistence Query Language），是一种针对实体的查询语言，无论是查询还是修改，全部操作的都是对象实体，而非数据库的表。

4.3.2 JPA 依赖配置

新建项目，在 pom 文件中加入 JPA 依赖、MySQL 依赖以及 Web 功能依赖，如代码清单 4-30 所示。

代码清单 4-30　JPA 项目依赖代码

```
<!-- JPA 依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- WEB 依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- Mysql 依赖-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

4.3.3　配置文件

在配置文件中加入数据库配置，与 4.1 节介绍的一致。接下来进行 JPA 的基本配置，比如 spring.jpa.hibernate.ddl-auto。其提供了如下几种配置。

- validate：在加载 hibernate 时，验证创建数据库表结构。
- create：每次加载 hibernate，重新创建数据库表结构，设置时要注意，如果设置错误的话，就会造成数据的丢失。
- create-drop：在加载的时候创建表，在关闭项目时删除表结构。
- update：加载时更新表结构。
- none：加载时不做任何操作。

根据具体情况选择配置即可。另外，如果需要，我们也可以加入 spring.jpa.show-sql 配置，设置为 true 时，可以在控制台打印 SQL。

案例配置代码如代码清单 4-31 所示。

代码清单 4-31　JPA 项目配置文件代码

```
##数据库配置
##数据库地址
spring.datasource.url=jdbc:mysql://localhost:3306/test?characterEncoding=
utf8&useSSL=false
##数据库用户名
spring.datasource.username=root
##数据库密码
spring.datasource.password=root
##数据库驱动
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

## JPA 配置
##validate  加载 hibernate 时，验证创建数据库表结构
##create  每次加载 hibernate，重新创建数据库表结构，这就是导致数据库表数据丢失的原因
##create-drop  加载 hibernate 时创建，退出时删除表结构
##update  加载 hibernate 自动更新数据库结构
##none  启动时不做任何操作
spring.jpa.hibernate.ddl-auto=create
##控制台打印 SQL
spring.jpa.show-sql=true
```

4.3.4　创建实体对象

创建一个实体对象，在类上加入注解@Entity 来表明这是一个实体类，在属性上使用@Id 表明

这是数据库中的主键 ID，使用@GeneratedValue(strategy = GenerationType.IDENTITY)表明此字段自增长，在属性上加入@Column(nullable = false,unique = true)可以设置字段的一些属性，比如 nullable 为非空、unique 唯一约束，还提供了其他属性，这里就不一一介绍了。User 实体类代码如代码清单 4-32 所示。

代码清单 4-32　JPA 项目 User 实体类代码

```
@Entity
public class User{

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY
)
    private Long id;
    @Column(nullable = false,unique = true)
    private String userName;
    @Column
    private String userPassword;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getUserPassword() {
        return userPassword;
    }

    public void setUserPassword(String userPassword) {
        this.userPassword = userPassword;
    }

    public User(Long id,String userName, String userPassword) {
        this.id = id;
        this.userName = userName;
        this.userPassword = userPassword;
    }
```

```
    public User(String userName, String userPassword) {
        this.userName = userName;
        this.userPassword = userPassword;
    }

    public User() {
    }
}
```

4.3.5　创建数据操作层

新建一个 repository 接口，使其继承 JpaRepository，这个接口默认提供一组与 JPA 规范相关的方法，其源代码如代码清单 4-33 所示。

代码清单 4-33　JPA 项目 JpaRepository 类代码

```
@NoRepositoryBean
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T,
ID>, QueryByExampleExecutor<T> {
    List<T> findAll();

    List<T> findAll(Sort var1);

    List<T> findAllById(Iterable<ID> var1);

    <S extends T> List<S> saveAll(Iterable<S> var1);

    void flush();

    <S extends T> S saveAndFlush(S var1);

    void deleteInBatch(Iterable<T> var1);

    void deleteAllInBatch();

    T getOne(ID var1);

    <S extends T> List<S> findAll(Example<S> var1);

    <S extends T> List<S> findAll(Example<S> var1, Sort var2);
}
```

从源代码中可以看到，默认为我们提供了很多简单的方法，如 findAll()、getOne()等，而 JpaRepository 则继承了 PagingAndSortingRepository 接口。PagingAndSortingRepository 接口代码如代码清单 4-34 所示。

代码清单 4-34　JPA 项目 PagingAndSortingRepository 类代码

```
@NoRepositoryBean
```

```
    public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {
        Iterable<T> findAll(Sort var1);

        Page<T> findAll(Pageable var1);
    }
```

PagingAndSortingRepository 接口继承了 CrudRepository 接口，实现了有关分页排序等相关的方法，其代码如代码清单 4-35 所示。

代码清单 4-35　JPA 项目 CrudRepository 类代码

```
    @NoRepositoryBean
    public interface CrudRepository<T, ID> extends Repository<T, ID> {
        <S extends T> S save(S var1);

        <S extends T> Iterable<S> saveAll(Iterable<S> var1);

        Optional<T> findById(ID var1);

        boolean existsById(ID var1);

        Iterable<T> findAll();

        Iterable<T> findAllById(Iterable<ID> var1);

        long count();

        void deleteById(ID var1);

        void delete(T var1);

        void deleteAll(Iterable<? extends T> var1);

        void deleteAll();
    }
```

CrudRepository 接口继承了 Spring Data JPA 的核心接口 Repository，实现了有关 CRUD 相关的方法（增、删、改、查）。在 Repository 接口中没有提供任何方法，仅仅作为一个标识来让其他类实现它作为仓库接口类，其代码如代码清单 4-36 所示。

代码清单 4-36　JPA 项目 Repository 类代码

```
    @Indexed
    public interface Repository<T, ID> {
    }
```

细心的读者可以看到，除了 Repository 接口以外，其余接口都含有一个@NoRepositoryBean 注解，加入这个注解的类，Spring 就不会实例化，用作父类的 Repository。

4.3.6　简单测试运行

新建一个 Controller 进行测试，创建一个 UserController，在控制层注入刚刚创建的 UserRepository，并在控制层创建 CURD 的 4 个方法，代码如代码清单 4-37 所示。

代码清单 4-37　JPA 项目 UserController 类代码

```
@RestController
@RequestMapping("test")
public class UserController {

    @Autowired
    private UserRepository userRepository;

    //http://localhost:8080/test/saveUser?userName=
%E5%A4%A7%E8%80%81%E6%9D%A8&userPassword=123
    @GetMapping(value = "/saveUser")
    public void saveUser(String userName,String userPassword){
        User user = new User(userName, userPassword);
        userRepository.save(user);
    }

    //http://localhost:8080/updateUser?Id=1&userName=
%E5%A4%A7%E8%80%81%E6%9D%A8&userPassword=1111
    @GetMapping(value = "/updateUser")
    public void updateUser(Long Id,String userName,String userPassword){
        User user = new User(Id,userName, userPassword);
        userRepository.save(user);
    }

    //http://localhost:8080/deleteUser?Id=1
    @GetMapping(value = "/deleteUser")
    public void deleteUser(Long Id){
        userRepository.deleteById(Id);
    }

    //http://localhost:8080/getUserById?Id=1
    @GetMapping(value = "/getUserById")
    public Optional<User> getUserById(Long Id){
        return userRepository.findById(Id);
    }
}
```

从上面的代码可以看到，在使用 JPA 操作数据库时，操作特别简单，基本上使用 Repository 提供的几个方法已经可以满足我们的需求。

4.3.7 JPA 扩展学习

前面学习了 JPA 的简单使用，接下来我们对它进行扩展学习。在 JPA 使用中，可以通过一些特定的命名规则实现对 SQL 语句条件的改变。

假设有一个 User 表，内含字段 id(int)、user_name(varchar[50])、pass_word(varchar[50])、age(int)、birthday(date)、is_enable(tinyint[1])。表 4-1 显示了通过特定命名规则对 SQL 语句的改变。

表 4-1 通过特定命名规则对表 User 进行的 SQL 语言的运用

序　号	关 键 字	举　例	范例 SQL
1	And	findByUserNameAndPassword	SELECT * FROM User WHERE user_name = ?1 AND pass_word = ?2
2	Or	findByUserNameOrPassword	SELECT * FROM User WHERE user_name = ?1 OR pass_word = ?2
3	Is	findByUserNameIs	SELECT * FROM User WHERE user_name = ?1
4	Equals	findByUserNameEquals	SELECT * FROM User WHERE user_name = ?1
5	Between	findByBirthdayBetween	SELECT * FROM User WHERE birthday BETWEEN ?1 AND ?2
6	LessThan	findByAgeLessThan	SELECT * FROM User WHERE age < ?1
7	LessThanEqual	findByAgeLessThanEqual	SELECT * FROM User WHERE age <= ?1
8	GreaterThan	findByAgeGreaterThan	SELECT * FROM User WHERE age > ?1
9	GreaterThanEqual	findByAgeGreaterThanEqual	SELECT * FROM User WHERE age >= ?1
10	After	findByBirthdayAfter	SELECT * FROM User WHERE birthday > ?1
11	Before	findByBirthdayBefor	SELECT * FROM User WHERE birthday < ?1
12	IsNull	findByPasswordIsNull	SELECT * FROM User WHERE pass_word IS NULL
13	IsNotNull	findByPasswordIsNotNull	SELECT * FROM User WHERE pass_word IS NOT NULL
14	NotNull	findByPasswordNotNull	SELECT * FROM User WHERE pass_word IS NOT NULL
15	Like	findByUserNameLike	SELECT * FROM User WHERE user_name LIKE ?1
16	NotLike	findByUserNameNotLike	SELECT * FROM User WHERE user_name NOT LIKE ?1
17	StartingWith	findByUserNameStartingWith	SELECT * FROM User WHERE user_name LIKE ?1 (parameter bound with appended %)

（续表）

序　　号	关　键　字	举　　例	范例 SQL
18	EndingWith	findByUserNameEndingWith	SELECT * FROM User WHERE user_name LIKE ?1 (parameter bound with prepended %)
19	Containing	findByUserNameContaining	SELECT * FROM User WHERE user_name LIKE ?1 (parameter bound wrapped in %)
20	OrderBy	findByUserNameOrderByAgeAsc	SELECT * FROM User WHERE user_name = ?1 ORDER BY age ASC
21	Not	findByUserNameNot	SELECT * FROM User WHERE user_name <> ?1
22	In	findByUserNameIn(Collection <Age> age)	SELECT * FROM User WHERE age IN ?1
23	NotIn	findByUserNameNotIn(Collection <Age> age)	SELECT * FROM User WHERE age NOT IN ?1
24	True	findByIsenableTrue	SELECT * FROM User WHERE userName = ?1 AND passWord = ?2
23	NotIn	findByUserNameNotIn(Collection <Age> age)	SELECT * FROM User WHERE age NOT IN ?1
24	True	findByIsenableTrue	SELECT * FROM User WHERE userName = ?1 AND passWord = ?2

上述方法都是基于既定的接口规则，其实 JPA 是支持注解形式执行 SQL 语句操作的，比如在接口上使用@Query，在内容中放入需要执行的 SQL 语句，如代码清单 4-38 所示。

代码清单 4-38　JPA 项目 findAllByUserName 方法代码

```
@Query("SELECT u FROM User u WHERE user_name = :userName")
User findAllByUserName(@Param("userName") String userName);
```

需要注意的是，这里是以对象为单位查询的，比如上面使用的是查询的 SELECT u，而不是我们在 SQL 内写的 SELECT *，使用@Param 给参数取别名，方便在 SQL 内使用。JPA 的使用就扩展到这里，其用法还有很多，感兴趣的读者可以参考官方文档：https://docs.spring.io/spring-data/data-jpa/docs/current/api/。

4.3.8　基于 WebFlux 的使用

之前我们提到过 Spring Boot 2.X 版本新提供的 WebFlux，接下来改用 WebFlux 操作 JPA。WebFlux 暂时还不支持关系型数据库，所以本小节的数据库改用 MongoDB，介绍响应式编程操作数据库。

1. 更换为 WebFlux 依赖

新建一个项目，在 pom 文件中加入 spring-boot-starter-data-mongodb-reactive 和 spring-boot-

starter-webflux 依赖。完整 pom 文件依赖代码内容如代码清单 4-39 所示。

代码清单 4-39　JPA 项目基于 WebFlux 依赖的代码

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-devtools</artifactId>
   <scope>runtime</scope>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
```

在配置文件中配置 MongoDB 数据库信息，配置内容如代码清单 4-40 所示。

代码清单 4-40　JPA 项目基于 WebFlux 配置文件代码

```
##mongo 配置
spring.data.mongodb.host=127.0.0.1
spring.data.mongodb.port=27017
spring.data.mongodb.database=test
```

创建一个实体类，实体类内容如代码清单 4-41 所示。

代码清单 4-41　JPA 项目基于 WebFlux 依赖 UserInfo 类代码

```
public class UserInfo {
    @Id
    private Long id;
    private String username;
    private String password;

    //省略 set、get 方法

    ...
}
```

创建数据操作层 UserRepository，在使用 WebFlux 时，我们需要继承 ReactiveMongoRepository，类代码内容如代码清单 4-42 所示。

代码清单 4-42　JPA 项目基于 WebFlux 依赖 UserRepository 类代码

```
public interface UserRepository extends
ReactiveMongoRepository<UserInfo,Long> {
}
```

在第 3 章我们学习 WebFlux 响应式编程的时候了解到，需要创建一个 Handler 用于实现具体方法，再创建一个 Router 用于路由跳转。首先，创建一个 UserHandler，我们先写一个保存用户的方法，在 WebFlux 中可以利用下面的方法直接获取 Post 请求对象体（Requestbody）中的对象，如代码清单 4-43 所示。

代码清单 4-43　JPA 项目基于 WebFlux 依赖 bodyToMono 的使用代码

```
Mono<UserInfo> user = request.bodyToMono(UserInfo.class);
```

其中，request 是 ServerRequest 对象，通过上面的方法可以直接将 Post 请求中的方法体（Requestbody）数据转换为 UserInfo 对象。我们来完善一下保存方法，方法内容如代码清单 4-44 所示。

代码清单 4-44　JPA 项目基于 WebFlux 依赖 saveUser 方法的代码

```
public Mono<ServerResponse> saveUser(ServerRequest request) {
    Mono<UserInfo> user = request.bodyToMono(UserInfo.class);
    return ServerResponse.ok().build(repository.insert(user).then());
}
```

在方法返回值中使用 ServerResponse.ok()方法表明返回状态成功，在类似插入、修改等不需要返回值的方法后面加入 then()方法返回一个 Mono<Void>。

修改方法与插入方法类似，这里不再赘述。接下来我们来看一个查询方法，比如想获取 ID 为 1 的用户，就可以利用 request.pathVariable 方法获取路径中的参数，如代码清单 4-45 所示。

代码清单 4-45　JPA 项目基于 WebFlux 依赖获取链接参数代码

```
Long userId = Long.valueOf(request.pathVariable("id"));
```

我们可以清楚地看到，上述代码就是从路径中获取 id 的 Long 值。完善一下方法，将查询到的用户返回，方法内容如代码清单 4-46 所示。

代码清单 4-46　JPA 项目基于 WebFlux 依赖 getUser 方法代码

```
public Mono<ServerResponse> getUser(ServerRequest request) {
    Long userId = Long.valueOf(request.pathVariable("id"));
```

```
        Mono<UserInfo> userInfo = repository.findById(userId);
        return ServerResponse.ok().contentType(APPLICATION_JSON).
body(userInfo, UserInfo.class);
    }
```

与插入方法不同的是，在返回值上表明了 contentType，这里设置成了 application/json，并且利用 body 方法将查询内容返回。

这里还列举了删除方法和查询列表方法的内容，和前面介绍的方法类似，可以参考使用。UserHandler 类如代码清单 4-47 所示。

代码清单 4-47　JPA 项目基于 WebFlux 依赖 UserHandler 类代码

```
    @Component
    public class UserHandler {

        private final UserRepository repository;

        public UserHandler(UserRepository repository) {
            this.repository = repository;
        }

        //http://localhost:8080/saveUser
        public Mono<ServerResponse> saveUser(ServerRequest request) {
            Mono<UserInfo> user = request.bodyToMono(UserInfo.class);
            return ServerResponse.ok().build(repository.insert(user).then());
        }

        //http://localhost:8080/deleteUser/1
        public Mono<ServerResponse> deleteUser(ServerRequest request) {
            Long userId = Long.valueOf(request.pathVariable("id"));
            return ServerResponse.ok().build(repository.deleteById(userId).
then());
        }

        //http://localhost:8080/user/1
        public Mono<ServerResponse> getUser(ServerRequest request) {
            Long userId = Long.valueOf(request.pathVariable("id"));
            Mono<UserInfo> userInfo = repository.findById(userId);
            return ServerResponse.ok().contentType(APPLICATION_JSON).
body(userInfo, UserInfo.class);
        }

        //http://localhost:8080/listUser
```

```
        public Mono<ServerResponse> listUser(ServerRequest request) {
            Flux<UserInfo> userList = repository.findAll();
            return ServerResponse.ok().contentType(APPLICATION_JSON).
body(userList, UserInfo.class);
        }

    }
```

创建一个 UserRouter 作为路由，在其中配置 UserHandler 类内方法对应路由，如代码清单 4-48 所示。

代码清单 4-48　JPA 项目基于 WebFlux 依赖 UserRouter 类代码

```
    @Configuration
    public class UserRouter {

        @Bean
        public RouterFunction<ServerResponse> routeUser(UserHandler
userHandler) {
            return RouterFunctions
                    .route(RequestPredicates.GET("/listUser")
                                    .and(RequestPredicates.accept(MediaType.
                                         APPLICATION_JSON)),
                            userHandler::listUser)
                    .andRoute(RequestPredicates.GET("/user/{id}")
                                    .and(RequestPredicates.accept(MediaType.
                                    APPLICATION_JSON)),
                            userHandler::getUser)
                    .andRoute(RequestPredicates.GET("/deleteUser/{id}")
                                    .and(RequestPredicates.accept(MediaType.
                                     APPLICATION_JSON)),
                            userHandler::deleteUser)
                    .andRoute(RequestPredicates.POST("/saveUser")
                                    .and(RequestPredicates.accept(MediaType.
                                    APPLICATION_JSON)),
                            userHandler::saveUser);
        }
    }
```

关于测试这里就不继续介绍了，以上方法都是笔者亲测可用的，感兴趣的读者可以自行测试。

4.4　使用 MyBatis 操作数据库

4.4.1　MyBatis 简介

在 MyBatis 官网（官网地址：http://www.mybatis.org/mybatis-3/zh/index.html）上是这样介绍 MyBatis 的：MyBatis 是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs（Plain Old Java Objects，普通的 Java 对象）映射成数据库中的记录。

通俗地理解，MyBatis 最大的优点是：

- 可以手写 SQL，比较灵活，对于很多互联网公司、业务迭代速度快的公司或者业务复杂的项目，MyBatis 修改、维护等方面更加灵活。
- 从学习成本上来说，MyBatis 上手更加容易，基本上没有更多学习成本，这是很多公司选用 MyBatis 的理由。
- 从 SQL 优化方面来说，手写的 SQL 优化起来更加方便。

4.4.2　MyBatis 依赖配置

创建项目，在 pom 文件中加入 MyBatis 依赖和 MySQL 数据库依赖，代码如代码清单 4-49 所示。

代码清单 4-49　MyBatis 项目依赖代码

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

4.4.3　配置文件

在配置文件中需要配置数据库信息以及 MyBatis 配置。数据库配置这里就不介绍了，关于 MyBatis 主要需要配置以下几种。

- logging.level.com.dalaoyang.dao.UserMapper：日志的打印级别，这里的 com.dalaoyang.dao.UserMapper 是本文案例中 Mapper 的位置，实际项目应该配置对应 Mapper 的位置。
- mybatis.mapper-locations：Mapper 文件的存放位置。
- mybatis.check-config-location：MyBatis 配置是否开启。
- mybatis.config-location：MyBatis 配置文件位置，与 mybatis.check-config-location 配合使用。

本案例对上述内容都进行了配置，配置文件代码如代码清单 4-50 所示。

代码清单 4-50　MyBatis 项目配置文件代码

```
##检查 mybatis 配置是否存在，一般命名为 mybatis-config.xml
mybatis.check-config-location =true
##配置文件位置
mybatis.config-location=classpath:mybatis/mybatis-config.xml
## mapper xml 文件地址
mybatis.mapper-locations=classpath*:mapper/*Mapper.xml
##日志级别
logging.level.com.springboot.dao.UserMapper=debug

##数据库 url
spring.datasource.url=jdbc:mysql://localhost:3306/test?characterEncoding=
utf8&useSSL=false
##数据库用户名
spring.datasource.username=root
##数据库密码
spring.datasource.password=root
##数据库驱动
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

在 src/mian/resources/mybatis 下创建 mybatis-config.xml，这个文件是 MyBatis 的全局配置文件，包含以下几种类型的配置：

- properties（属性）
- settings（全局配置参数）
- typeAliases（类型别名）
- typeHandlers（类型处理器）
- objectFactory（对象工厂）
- plugins（插件）
- environments（环境集合属性对象）
- environment（环境子属性对象）
- transactionManager（事务管理）
- dataSource（数据源）
- mappers（映射器）

案例中仅配置了一些常使用的类型别名 typeAliases，Mybatis-config.xml 内容如代码清单 4-51 所示。

代码清单 4-51　MyBatis 项目 Mybatis-config 配置代码

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD SQL Map Config 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <typeAliases>
        <typeAlias alias="Integer" type="java.lang.Integer" />
        <typeAlias alias="Long" type="java.lang.Long" />
        <typeAlias alias="HashMap" type="java.util.HashMap" />
        <typeAlias alias="LinkedHashMap" type="java.util.LinkedHashMap" />
        <typeAlias alias="ArrayList" type="java.util.ArrayList" />
        <typeAlias alias="LinkedList" type="java.util.LinkedList" />
        <typeAlias alias="user" type="com.dalaoyang.entity.User"/>
    </typeAliases>
</configuration>
```

创建实体类 User，其中使用@Alias 注解也可以表明类别名，代码如代码清单 4-52 所示。

代码清单 4-52　MyBatis 项目 User 实体类代码

```
@Alias("user")
public class User {

    private int id;
    private String user_name;
    private String user_password;

    //省略 set、get 方法
    ...
}
```

4.4.4　基于 XML 的使用

创建 Mapper 对应接口类 UserMapper，在类上加入注解@Mapper，表明这是一个 Mapper。我们提前定义 5 个方法，分别是：

- 根据用户名查询用户。
- 根据用户名修改用户。
- 根据用户名删除用户。
- 保存用户。
- 获取用户列表。

UserMapper 接口代码如代码清单 4-53 所示。

代码清单 4-53　MyBatis 项目 UserMapper 类代码

```
@Mapper
public interface UserMapper {
    User findUserByUsername(String username);

    void updateUserByUsername(User user);

    void deleteUserByUsername(String username);

    void saveUser(User user);

    List<User> getUserList();
}
```

在 src/mian/resources/mapper 下创建 UserMapper.xml，对应写好在 UserMapper 接口类的方法，完整内容如代码清单 4-54 所示。

代码清单 4-54　MyBatis 项目 UserMapper.xml 配置代码

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.dalaoyang.dao.UserMapper">
    <resultMap id="user" type="com.dalaoyang.entity.User"/>
    <parameterMap id="user" type="com.dalaoyang.entity.User"/>
    <select id="findUserByUsername" parameterType="String"
resultMap="user">
        SELECT * FROM user
        WHERE user_name=#{1}
    </select>
    <update id="updateUserByUsername" parameterMap="user">
        UPDATE USER SET USER_PASSWORD=#{user_password} WHERE
USER_NAME=#{user_name}
    </update>
    <delete id="deleteUserByUsername" parameterType="String">
        DELETE FROM USER WHERE USER_NAME=#{1}
    </delete>
    <!-- 使用 alias 自定义的 parameterType-->
    <insert id="saveUser" parameterType="user">
        INSERT INTO USER (user_password,user_name) VALUES (#{user_password},
#{user_name})
    </insert>
    <select id="getUserList" resultMap="user">
```

```
        SELECT  * FROM USER
    </select>
</mapper>
```

因为 MyBatis 深受很多公司的喜爱，所以介绍一下 Mapper 的标签。标签大致分为以下几种。

（1）定义 SQL 语句

- insert：多用于执行插入语句，标签内有两个属性 id（唯一标识符）和 parameterType（传入的参数类型）。
- delete：多用于执行删除语句，标签内有两个属性 id（唯一标识符）和 parameterType（传入的参数类型）。
- update：多用于执行修改语句，标签内有两个属性 id（唯一标识符）和 parameterType（传入的参数类型）。
- select：用于执行查询，与上面三个标签相比，多了一个 resultType 属性，用于接收返回类型。

注　意
比如在 insert 标签内写 delete 语句不会报错，但是不建议这样使用。

（2）结果集

- resultMap：用于建立 SQL 查询结果字段与实体属性的映射关系信息。

（3）动态 SQL 拼接

- if：用于判断，在 test 属性内加入条件。
- choose：用于判断，与 when 和 otherwise 配合使用。
- foreach：循环语句，其中包含属性 collection（集合，内容可以是 list、array 和 map）、item（循环遍历的元素）、index（下标）、open（前缀）、close（后缀）、separator（分隔符）。

（4）格式化输出

- where：根据标签内的值是否存在自动拼接 where 语句。
- set：根据标签内的值是否存在自动拼接 set 语句。
- trim：多用于灵活去除多余关键字的标签，一般结合 where 或 set 使用。

（5）配置关联关系

- collection：用于配置一对一关系。
- association：用于配置一对多关系。

（6）SQL 标签

- sql：主要用于提取 sql 片段，便于复用。

4.4.5　基于注解使用

MyBatis 不仅可以使用 XML 形式操作数据库，还可以使用注解形式操作数据库，比如如下注解。

- @Select：其中值写查询 SQL。
- @Update：其中值写修改 SQL。
- @Delete：其中值写删除 SQL。
- @Insert：其中值写插入 SQL。
- @Results：是以@Result 为元素的数据。
- @Result：映射实体类属性和字段之间的关系。
- @ResultMap：用于解决返回结果映射问题，与上面介绍的 resultMap 标签功能类似。
- @Result：可以用作表明自定义对象，方便内容重用。
- @SelectProvider：相当于直接使用在类中写好的 SQL，将 SQL 封装到类内，方便管理。type 属性表明使用哪个类，method 对应使用方法。
- @UpdateProvider：功能类似于@SelectProvider。
- @DeleteProvider：功能类似于@SelectProvider。
- @InsertProvider：功能类似于@SelectProvider。

其实现效果和使用 XML 模式是一样的，并且两种模式可以混用。例子如代码清单 4-55 所示。

代码清单 4-55　MyBatis 项目基于注解的代码

```
@Results({
@Result(property = "id",column = "id"),
@Result(property = "user_name",column = "user_name"),
       @Result(property = "pass_word",column = "pass_word")
})
@Select("SELECT * FROM USER")
List<User> findAll();

@SelectProvider(type = UserSqlProvider.class,method = "getSql")
List<User> findUserById(@Param("id") int id);
```

4.4.6　测试运行

前面对大部分使用场景进行了介绍，接下来进行测试。新建一个 Controller，分别对刚刚写的每一个数据库操作写一个方法进行测试，代码内容如代码清单 4-56 所示。

代码清单 4-56　MyBatis 项目 UserController 类代码

```
@RestController
public class UserController {

    @Autowired
    private UserMapper userMapper;

    //http://localhost:8080/getUser?username=xiaoli2
    @GetMapping("/getUser")
    public String getUser(String username){
        User user =userMapper.findUserByUsername(username);
        return user!=null ? username+"的密码是: "+user.getUser_password():"不
存在用户名为"+username+"的用户";
    }

    //http://localhost:8080/updateUser?username=xiaoli2&password=123
    @GetMapping("/updateUser")
    public String updateUser(String password,String username){
        User user = new User(username,password);
        userMapper.updateUserByUsername(user);
        return "success!";
    }


    //http://localhost:8080/addUser?username=xiaoli2&password=123
    @GetMapping("/addUser")
    public String addUser(String username,String password){
        User user = new User(username,password);
        userMapper.saveUser(user);
        return "success!";
    }

    //http://localhost:8080/deleteUser?username=xiaoli2
    @GetMapping("/deleteUser")
    public String deleteUser(String username){
        userMapper.deleteUserByUsername(username);
        return "success!";
    }

    //http://localhost:8080/getUserList
    @GetMapping("/getUserList")
    public List getUserList(){
        return userMapper.getUserList();
```

```
    }

    //http://localhost:8080/findAll
    @GetMapping("/findAll")
    public List findAll(){
        return userMapper.findAll();
    }

    //http://localhost:8080/findUserById?id=1
    @GetMapping("/findUserById")
    public List findUserById(int id){
        return userMapper.findUserById(id);
    }
}
```

具体测试可以在浏览器上访问代码中的注释，每一个方法笔者都对应写了测试地址，以上都是笔者亲测无误的。

4.4.7　Mybatis-Generator 插件学习

由于业务的不断增长，数据库中的表也随之增长，造成在没创建表时就需要在项目内反复创建实体类、Mapper 文件、dao 层文件等，这样的重复工作虽然难度不大，但是会浪费人力，因此 MyBatis 创建了一个针对这个问题的插件 Mybatis-Generator。Mybatis-Generator 是 MyBatis 官方提供的一个便捷型插件，利用它可以根据数据库表结构自动在项目内创建对应的实体类、Mapper 文件和 dao 层。

（1）在 pom 文件中加入 Mybatis-Generator 插件

在 pom 文件中加入 Mybatis-Generator 插件，为了方便观看，这里展示完整 pom 文件代码，如代码清单 4-57 所示。

代码清单 4-57　Mybatis-Generator 项目依赖文件代码

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.dalaoyang</groupId>
    <artifactId>springboot_generator</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
```

```
    <name>chapter4.4.7</name>
    <description>chapter4.4.7</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8
</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>1.3.1</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
```

```
                <groupId>org.mybatis.generator</groupId>
                <artifactId>mybatis-generator-maven-plugin</artifactId>
                <version>1.3.2</version>
                <executions>
                    <execution>
                        <id>mybatis-generator</id>
                        <phase>deploy</phase>
                        <goals>
                            <goal>generate</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <!-- Mybatis-Generator 工具配置文件的位置 -->
                    <configurationFile>src/main/resources/mybatis-generator/
generatorConfig.xml</configurationFile>
                    <verbose>true</verbose>
                    <overwrite>true</overwrite>
                </configuration>
                <dependencies>
                    <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <version>5.1.46</version>
                    </dependency>
                    <dependency>
                        <groupId>org.mybatis.generator</groupId>
                        <artifactId>mybatis-generator-core</artifactId>
                        <version>1.3.2</version>
                    </dependency>
                </dependencies>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <classifier>exec</classifier>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

这里需要注意的是，configurationFile 配置的是 Mybatis-Generator 插件所存放的位置，本案例

是在 src/main/resources/mybatis-generator 下创建了一个 generatorConfig.xml 文件，在配置文件中对需要配置的内容做了详细的说明，配置内容如代码清单 4-58 所示。

代码清单 4-58　Mybatis-Generator 项目配置文件代码

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<!-- 配置生成器 -->
<generatorConfiguration>
    <!--执行 generator 插件生成文件的命令：call mvn mybatis-generator:generate -e
-->
    <!-- 引入配置文件 -->
    <properties resource="application.properties"/>
    <!--classPathEntry:数据库的 JDBC 驱动，换成你自己的驱动位置，可选 -->
    <!--<classPathEntry location="D:\generator_mybatis\mysql-connector-
java-5.1.24-bin.jar" /> -->

    <!-- 一个数据库一个 context -->
    <!--defaultModelType="flat" 大数据字段，不分表 -->
    <context id="MysqlTables" targetRuntime="MyBatis3Simple"
defaultModelType="flat">
        <!-- 自动识别数据库关键字，默认为 false，如果设置为 true，就根据
SqlReservedWords 中定义的关键字列表。一般保留默认值，遇到数据库关键字（Java 关键字）时，
使用 columnOverride 覆盖 -->
        <property name="autoDelimitKeywords" value="true" />
        <!-- 生成的 Java 文件的编码 -->
        <property name="javaFileEncoding" value="utf-8" />
        <!-- beginningDelimiter 和 endingDelimiter：指明数据库中用于标记数据库对象
名的符号，比如 ORACLE 就是双引号，MySQL 默认是`（反引号）-->
        <property name="beginningDelimiter" value="`" />
        <property name="endingDelimiter" value="`" />

        <!-- 格式化 java 代码 -->
        <property name="javaFormatter" value="org.mybatis.generator.
api.dom.DefaultJavaFormatter"/>
        <!-- 格式化 XML 代码 -->
        <property name="xmlFormatter" value="org.mybatis.generator.
api.dom.DefaultXmlFormatter"/>
        <plugin type="org.mybatis.generator.plugins.SerializablePlugin" />

        <plugin type="org.mybatis.generator.plugins.ToStringPlugin" />

        <!-- 注释 -->
```

```
        <commentGenerator >
            <property name="suppressAllComments" value="false"/><!-- 是否取消
注释 -->
            <property name="suppressDate" value="true" /> <!-- 是否生成注释带时
间戳-->
        </commentGenerator>

        <!-- jdbc 连接 -->
        <jdbcConnection driverClass="${spring.datasource.driver-class-name}"
connectionURL="${spring.datasource.url}" userId=
"${spring.datasource.username}" password="${spring.datasource.password}" />
        <!-- 类型转换 -->
        <javaTypeResolver>
            <!-- 是否使用 bigDecimal， false 可自动转化为其他类型（Long、Integer、
Short 等） -->
            <property name="forceBigDecimals" value="false"/>
        </javaTypeResolver>

        <!-- 生成实体类地址 -->
        <javaModelGenerator targetPackage="com.dalaoyang.entity"
targetProject="${mybatis.project}" >
            <property name="enableSubPackages" value="false"/>
            <property name="trimStrings" value="true"/>
        </javaModelGenerator>
        <!-- 生成 mapxml 文件 -->
        <sqlMapGenerator targetPackage="mapper" targetProject=
"${mybatis.resources}" >
            <property name="enableSubPackages" value="false" />
        </sqlMapGenerator>
        <!-- 生成 mapxml 对应 client，也就是接口 dao -->
        <javaClientGenerator targetPackage="com.dalaoyang.dao"
targetProject="${mybatis.project}" type="XMLMAPPER" >
            <property name="enableSubPackages" value="false" />
        </javaClientGenerator>
        <!-- table 可以有多个，每个数据库中的表都可以写一个 table，tableName 表示要匹
配的数据库表，也可以在 tableName 属性中通过使用%通配符来匹配所有数据库表，只有匹配的表才会
自动生成文件 -->
        <table tableName="user" enableCountByExample="true"
enableUpdateByExample="true" enableDeleteByExample="true"
enableSelectByExample="true" selectByExampleQueryId="true">
            <property name="useActualColumnNames" value="false" />
            <!-- 数据库表主键 -->
            <generatedKey column="id" sqlStatement="Mysql" identity="true" />
        </table>
```

```
    </context>
</generatorConfiguration>
```

在 Mybatis-Generator 中有部分数据这里读取的是 application.properties 文件中的内容。下面给出 application.properties 文件中的配置，如代码清单 4-59 所示。

代码清单 4-59　Mybatis-Generator 项目配置文件代码

```
## mapper xml 文件地址
mybatis.mapper-locations=classpath*:mapper/*Mapper.xml

##数据库 url
spring.datasource.url=jdbc:mysql://localhost:3306/test?characterEncoding=
utf8&useSSL=false
##数据库用户名
spring.datasource.username=root
##数据库密码
spring.datasource.password=123456
##数据库驱动
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

#Mybatis Generator configuration
#dao 类和实体类的位置
mybatis.project =src/main/java
#mapper 文件的位置
mybatis.resources=src/main/resources
```

到这里，其实就已经配置完成了。接下来我们查看 IntelliJ IDEA 中，Maven 的工具栏，可以看到已经安装了 Mybatis-Generator 插件，如图 4-15 所示。

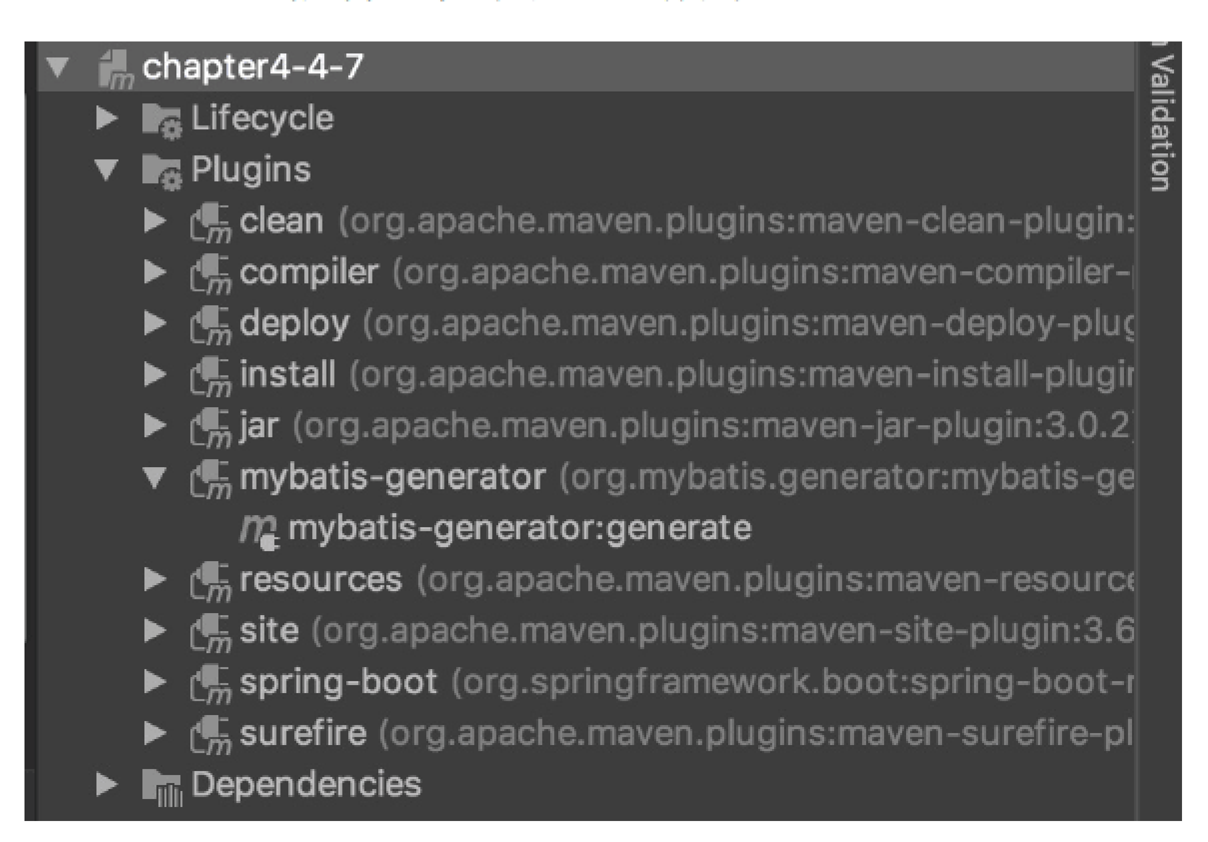


图 4-15　Mybatis-Generator 项目插件示例图

我们看一下 Maven 操作日志，提示已经生成了 dao 层、实体类、Mapper 文件，如图 4-16 所示。去对应目录看看，果然这些都生成了，如图 4-17 所示。

```
chapter4-4-7 [org.mybatis.generator:mybatis-generator-maven-plugin:1.3.2:generate]
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java -Dmaven.multiModuleProjectDirectory=/Users/dalaoyang/Demo/SpringBoot_Book/chapter4-4-7 -Dmav
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building chapter4-4-7 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- mybatis-generator-maven-plugin:1.3.2:generate (default-cli) @ chapter4-4-7 ---
[INFO] Connecting to the Database
log4j:WARN No appenders could be found for logger (org.mybatis.generator.internal.db.DatabaseIntrospector).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
[INFO] Introspecting table user
[INFO] Generating Record class for table user
[INFO] Generating Mapper Interface for table user
[INFO] Generating SQL Map for table user
[INFO] Saving file UserMapper.xml
[INFO] Saving file User.java
[INFO] Saving file UserMapper.java
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.596 s
[INFO] Finished at: 2018-09-16T14:56:14+08:00
[INFO] Final Memory: 13M/171M
[INFO] ------------------------------------------------------------------------

Process finished with exit code 0
```

图 4-16　Mybatis-Generator 项目执行 LOG 示例图

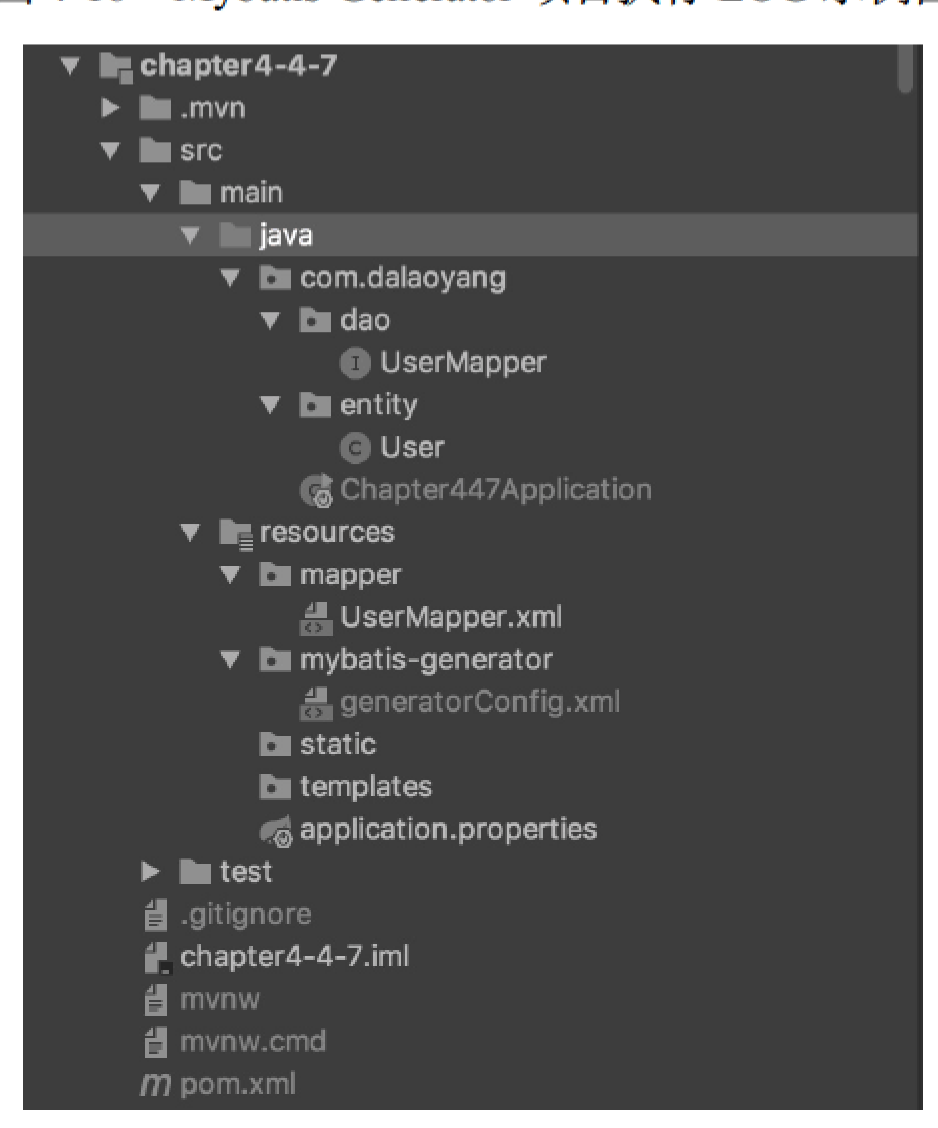


图 4-17　Mybatis-Generator 项目结构示例图

我们再来查看一下 Mapper，其实在 Mapper 内已经默认生成了几个简单的方法，让我们看一下 UserMapper 接口的内容，代码如代码清单 4-60 所示。

代码清单 4-60　Mybatis-Generator 项目 UserMapper 类代码

```
public interface UserMapper {
    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table user
     *
     * @mbggenerated
     */
    int deleteByPrimaryKey(Long id);
```

```
    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table user
     *
     * @mbggenerated
     */
    int insert(User record);

    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table user
     *
     * @mbggenerated
     */
    User selectByPrimaryKey(Long id);

    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table user
     *
     * @mbggenerated
     */
    List<User> selectAll();

    /**
     * This method was generated by MyBatis Generator.
     * This method corresponds to the database table user
     *
     * @mbggenerated
     */
    int updateByPrimaryKey(User record);
}
```

UserMapper.xml 也对应写好了 SQL，代码内容如代码清单 4-61 所示。

代码清单 4-61　Mybatis-Generator 项目 UserMapper.xml 代码

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.dalaoyang.dao.UserMapper" >
  <resultMap id="BaseResultMap" type="com.dalaoyang.entity.User" >
    <!--
      WARNING - @mbggenerated
      This element is automatically generated by MyBatis Generator, do not
modify.
```

```
        -->
        <id column="id" property="id" jdbcType="BIGINT" />
        <result column="user_name" property="userName" jdbcType="VARCHAR" />
        <result column="user_password" property="userPassword"
jdbcType="VARCHAR" />
      </resultMap>
      <delete id="deleteByPrimaryKey" parameterType="java.lang.Long" >
        <!--
          WARNING - @mbggenerated
          This element is automatically generated by MyBatis Generator, do not
modify.
        -->
        delete from user
        where id = #{id,jdbcType=BIGINT}
      </delete>
      <insert id="insert" parameterType="com.dalaoyang.entity.User" >
        <!--
          WARNING - @mbggenerated
          This element is automatically generated by MyBatis Generator, do not
modify.
        -->
        <selectKey resultType="java.lang.Long" keyProperty="id" order="AFTER" >
          SELECT LAST_INSERT_ID()
        </selectKey>
        insert into user (user_name, user_password)
        values (#{userName,jdbcType=VARCHAR}, #{userPassword,jdbcType=VARCHAR})
      </insert>
      <update id="updateByPrimaryKey"
parameterType="com.dalaoyang.entity.User" >
        <!--
          WARNING - @mbggenerated
          This element is automatically generated by MyBatis Generator, do not
modify.
        -->
        update user
        set user_name = #{userName,jdbcType=VARCHAR},
          user_password = #{userPassword,jdbcType=VARCHAR}
        where id = #{id,jdbcType=BIGINT}
      </update>
      <select id="selectByPrimaryKey" resultMap="BaseResultMap" parameterType=
"java.lang.Long" >
        <!--
          WARNING - @mbggenerated
```

```
        This element is automatically generated by MyBatis Generator, do not
modify.
       -->
       select id, user_name, user_password
       from user
       where id = #{id,jdbcType=BIGINT}
     </select>
     <select id="selectAll" resultMap="BaseResultMap" >
       <!--
         WARNING - @mbggenerated
         This element is automatically generated by MyBatis Generator, do not
modify.
       -->
       select id, user_name, user_password
       from user
     </select>
   </mapper>
```

这些自动生成的方法都是可以直接使用的，可以使用在 Controller 内对应的写方法测试，也可以使用测试用例测试，这里就不一一测试了。

4.4.8 PageHelper 插件

在操作数据库的时候，分页是必不可少的一项任务，在使用 MyBatis 时，可以利用 PageHelper 插件对 MyBatis 插件进行分页。PageHelper 支持常见的 12 种数据库，如 Oracle、MySQL、MariaDB、SQLite、DB2、PostgreSQL、SQL Server 等。

在 pom 文件中加入 PageHelper 依赖，依赖代码如代码清单 4-62 所示。

代码清单 4-62 Mybatis-PageHelper 项目依赖文件代码

```
<!--pagehelper -->
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper-spring-boot-starter</artifactId>
    <version>1.2.5</version>
</dependency>
```

配置文件除了 MySQL 配置外，还添加了一个 PageHelper 插件的配置，也就是配置 SQL 方言，配置如代码清单 4-63 所示。

代码清单 4-63 Mybatis-PageHelper 项目配置文件代码

```
#pagehelper 分页插件配置
pagehelper.helperDialect=mysql
```

实体类和之前的一样，这里就不反复介绍了。由于 PageHelper 插件只是用于分页，因此我们只是在 Mapper 内用注解写了一个查询所有用户的方法，如代码清单 4-64 所示。

代码清单 4-64　Mybatis-PageHelper 项目 UserMapper 类代码

```
@Mapper
public interface UserMapper {

    @Select("SELECT * FROM USER")
    List<User> getUserListPage();
}
```

使用 PageHelper 其实特别方便，只要引入 PageHelper 类，然后设置页码和每页的数量即可，案例测试代码如代码清单 4-65 所示。

代码清单 4-65　Mybatis-PageHelper 项目 UserController 类代码

```
@RestController
public class UserController {

    @Autowired
    private UserMapper userMapper;

    //http://localhost:8080/getUserListPage?pageNum=1&pageSize=2
    @GetMapping("getUserListPage")
    public List<User> getUserListPage(Integer pageNum, Integer pageSize){
        PageHelper.startPage(pageNum, pageSize);
        return userMapper.getUserListPage();
    }
}
```

启动项目，在浏览器上访问 http://localhost:8080/getUserListPage?pageNum=1&pageSize=2，这里 pageNum 为页码、pageSize 为每页的数量。是不是很简单？接下来我们介绍一个更全面的插件。

4.4.9　Mybatis-Plus 插件

Mybatis-Plus 是苞米豆团队开发的一个 MyBatis 增强型插件，官网上介绍只做增强，不做改变，为简化开发、提高效率而生。其特性有很多，这里不一一介绍，感兴趣的读者可以在官网上查看，官网地址：https://mp.baomidou.com/。

接下来我们学习 Spring Boot 如何使用 Mybatis-Plus 插件。新建项目，在 pom 文件中加入 Mybatis-Plus 依赖，完整 pom 文件依赖如代码清单 4-66 所示。

代码清单 4-66　Mybatis-Plus 项目依赖文件代码

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatisplus-spring-boot-starter</artifactId>
        <version>1.0.5</version>
    </dependency>
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus</artifactId>
        <version>2.3</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

接下来在配置文件中配置 mapper.xml 文件的位置和 type-aliases 实体的位置，配置如代码清单 4-67 所示。

代码清单 4-67　Mybatis-Plus 项目配置文件代码

```
##mybatis-plus mapper xml 文件地址
mybatis-plus.mapper-locations=classpath*:mapper/*Mapper.xml
##mybatis-plus type-aliases 文件地址
mybatis-plus.type-aliases-package=com.dalaoyang.entity
```

实体类还是使用之前的 User 类，可以复制过来，新建一个 Mybatis-Plus 配置类 MybatisPlusConfig，在这里可以设置一些方言的配置等，代码如代码清单 4-68 所示。

代码清单 4-68　Mybatis-Plus 项目 MybatisPlusConfig 类代码

```
@Configuration
```

```
public class MybatisPlusConfig {
    @Bean
    public PaginationInterceptor paginationInterceptor(){
        PaginationInterceptor page = new PaginationInterceptor();
        //设置方言类型
        page.setDialectType("mysql");
        return page;
    }
}
```

关于 dao 层，这里我们需要继承 Mybatis-Plus 提供的 BaseMapper，只在里面写一个查询用户列表（getUserList）的方法，代码如代码清单 4-69 所示。

代码清单 4-69　Mybatis-Plus 项目 UserMapper 类代码

```
@Mapper
public interface UserMapper extends BaseMapper<User> {
    List<User> getUserList();
}
```

在 Mapper.xml 内写出对应方法，如代码清单 4-70 所示。

代码清单 4-70　Mybatis-Plus 项目 Mapper.xml 代码

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.dalaoyang.dao.UserMapper">
    <resultMap id="user" type="com.dalaoyang.entity.User"/>
    <parameterMap id="user" type="com.dalaoyang.entity.User"/>

    <select id="getUserList" resultMap="user">
        SELECT  * FROM USER
    </select>
</mapper>
```

最后使用 Controller 进行测试，方法介绍如下。

- getUserList：尝试使用传统 MyBatis 方法，若可以使用，则证明 Mybatis-Plus 插件没有影响原有使用。
- getUserListByName：使用 Mybatis-Plus 提供的 selectByMap 方法，参数是 Map，在 Map 内写入查询条件。
- saveUser：使用 Mybatis-Plus 提供的 insert 方法，参数使用对应实体即可，返回影响行数。
- updateUser：使用 Mybatis-Plus 提供的 insert 方法，参数使用对应实体（带 ID）即可，返回影响行数。

- getUserListByPage：使用 Mybatis-Plus 提供的 selectPage 方法，这是一个条件分页查询方法，需要用到 Mybatis-Plus 的对象 EntityWrapper 存放查询条件，使用 Page 来存放分页信息。

关于测试 Mybatis-Plus 的完整 Controller 代码如代码清单 4-71 所示。

代码清单 4-71　Mybatis-Plus 项目依赖文件代码

```
@RestController
public class UserController {
    @Autowired
    private UserMapper userDao;

    //http://localhost:8080/getUserList
    @GetMapping("getUserList")
    public List<User> getUserList(){
        return userDao.getUserList();
    }

    //http://localhost:8080/getUserListByName?userName=xiaoli
    //条件查询
    @GetMapping("getUserListByName")
    public List<User> getUserListByName(String userName)
    {
        Map map = new HashMap();
        map.put("user_name", userName);
        return userDao.selectByMap(map);
    }

    //http://localhost:8080/saveUser?userName=xiaoli&userPassword=111
    //保存用户
    @GetMapping("saveUser")
    public String saveUser(String userName,String userPassword)
    {
        User user = new User(userName,userPassword);
        Integer index = userDao.insert(user);
        if(index>0){
            return "新增用户成功。";
        }else{
            return "新增用户失败。";
        }
    }

    //http://localhost:8080/updateUser?id=5&userName=xiaoli&userPassword=111
    //修改用户
```

```
    @GetMapping("updateUser")
    public String updateUser(Integer id,String userName,String userPassword)
    {
        User user = new User(id,userName,userPassword);
        Integer index = userDao.updateById(user);
        if(index>0){
            return "修改用户成功，影响行数"+index+"行。";
        }else{
            return "修改用户失败，影响行数"+index+"行。";
        }
    }

    //http://localhost:8080/getUserById?userId=1
    //根据 Id 查询 User
    @GetMapping("getUserById")
    public User getUserById(Integer userId)
    {
        return userDao.selectById(userId);
    }

    //http://localhost:8080/getUserListByPage?pageNumber=1&pageSize=2
    //条件分页查询
    @GetMapping("getUserListByPage")
    public List<User> getUserListByPage(Integer pageNumber,Integer pageSize)
    {
        Page<User> page =new Page<>(pageNumber,pageSize);
        EntityWrapper<User> entityWrapper = new EntityWrapper<>();
        entityWrapper.eq("user_name", "xiaoli");
        return userDao.selectPage(page,entityWrapper);
    }

}
```

4.5　配置多数据源

什么是数据源？

数据源是提供某种所需要数据的器件或原始媒体。在数据源中存储了所有建立数据库连接的信息。就像通过指定文件名称可以在文件系统中找到文件一样，通过提供正确的数据源名称可以找到相应的数据库连接。本节将对多数据源进行学习。

4.5.1 多数据源情况分析

什么是多数据源？从字面意思来看，其实就是使用多个数据源方法的多个数据库，数据源和数据库是一对一的关系。这时，我们会遇到另一个问题，为什么不用一个数据库来解决问题，而要分那么多数据源呢？接下来我们一起来探讨。

1. 为什么要使用多数据源

这个问题其实和为什么要分库的中心思想是一致的。随着业务的发展，数据库中的数据量和数据表越来越多，对数据库的操作（增、删、改、查）开销越来越大，最后可能导致所在服务器的资源被占用，如服务器 CPU 被严重占用。因此，多数据源出现了，将数据库分成多个，将压力分散开来，减小数据库的压力。

2. 如何分多数据源

一般情况下，分数据源大致有两种：

- 垂直切分。所谓垂直切分，我们可以理解为根据业务功能分成多个数据库。以商城系统为例，有商品模块、用户模块、订单模块，可能还有记录日志的数据表等，以上几个模块可以分为 GoodsDb（商品数据库）、UserDb（用户数据库）、OrderDb（订单数据库）、LogDb（日志数据库）。
- 水平切分。水平切分一般来说是根据数据量来分的，比如一个数据库的数据量特别大的话，我们可以按某种规则将数据库分为多个，比如按年份，年份较近的优先对待。

3. 多数据源的好处是什么

多数据源有如下好处：

- 数据逻辑清晰，维护方便，对应数据源所包含的数据都是同业务类型的。
- 减小数据库的压力，降低服务器的负载，让原来同一台机器的压力分散开来。

4.5.2 配置多数据源

前面介绍了使用多数据源的原因，接下来我们一起来学习 Spring Boot 是如何使用多数据源的。

首先介绍有关多数据源的配置，比如我们有两个数据源：test 和 test2，需要在配置文件中进行数据库对应的配置，如代码清单 4-72 所示。

代码清单 4-72 多数据源项目配置文件代码

```
## test 数据源
##数据库 url
spring.datasource.test.jdbc-url=jdbc:mysql://localhost:3306/test?characterEncoding=utf8&useSSL=false
```

```
##数据库用户名
spring.datasource.test.username=root
##数据库密码
spring.datasource.test.password=root
##数据库驱动
spring.datasource.test.driver-class-name=com.mysql.jdbc.Driver

## test2 数据源
##数据库 url
spring.datasource.test2.jdbc-url=jdbc:mysql://localhost:3306/test2?charac
terEncoding=utf8&useSSL=false
##数据库用户名
spring.datasource.test2.username=root
##数据库密码
spring.datasource.test2.password=root
##数据库驱动
spring.datasource.test2.driver-class-name=com.mysql.jdbc.Driver
```

注意多数据源与单数据源配置的不同，之前单数据源数据库是直接配置 spring.datasource.password，这次在 spring.datasource 后面加了一个数据源名称。

配置好数据源之后，新建一个数据源配置类，用于接收刚刚的配置参数，这里需要使用@Primary 注解表明哪个是主数据源，内容如代码清单 4-73 所示。

代码清单 4-73　多数据源项目 DataSourceConfig 类代码

```
@Configuration
public class DataSourceConfig {
    @Bean(name = "testDataSource")
    @Qualifier("testDataSource")
    @Primary
    @ConfigurationProperties(prefix="spring.datasource.test")
    public DataSource primaryDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "test2DataSource")
    @Qualifier("test2DataSource")
    @ConfigurationProperties(prefix="spring.datasource.test2")
    public DataSource secondaryDataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

接下来需要对每个数据源进行详细配置。这里以 test1 数据源为例，其中详细配置当前数据源的实体类位置、数据库信息等，如代码清单 4-74 所示。

代码清单 4-74　多数据源项目 TestDataSourceConfig 类代码

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
        entityManagerFactoryRef="entityManagerFactoryPrimary",
        transactionManagerRef="transactionManagerPrimary",
        basePackages= { "com.springboot.repository.datasource" })
public class TestDataSourceConfig {
    @Autowired
    @Qualifier("testDataSource")
    private DataSource dataSource;

    @Primary
    @Bean(name = "entityManagerPrimary")
    public EntityManager entityManager(EntityManagerFactoryBuilder builder){
        return entityManagerFactoryPrimary(builder).getObject().
createEntityManager();
    }

    @Primary
    @Bean(name = "entityManagerFactoryPrimary")
    public LocalContainerEntityManagerFactoryBean entityManagerFactoryPrimary
(EntityManagerFactoryBuilder builder) {
        return builder
                .dataSource(dataSource)
                .properties(getVendorProperties())
                .packages("com.springboot.entity.datasource")
                //设置实体类所在位置
                .persistenceUnit("primaryPersistenceUnit")
                .build();
    }

    @Autowired
    private JpaProperties jpaProperties;

    private Map<String, Object> getVendorProperties() {
        return jpaProperties.getHibernateProperties(new HibernateSettings());
    }

    @Primary
    @Bean(name = "transactionManagerPrimary")
    public PlatformTransactionManager transactionManagerPrimary
(EntityManagerFactoryBuilder builder) {
```

```
        return new JpaTransactionManager(entityManagerFactoryPrimary
(builder).getObject());
    }
}
```

4.5.3　基于 JPA 使用多数据源

有关 JPA 的使用之前已经介绍很多了，接下来使用 JPA 测试前面的多数据源是否生效。在 pom 文件加入依赖，分别创建两个实体类，其中 City 对应数据源 test 的对应实体类位置，House 对应数据源 test2 的对应实体类位置。启动项目，观察控制台和数据库，可以看到在 test 数据库中创建了 City 表，在 City 数据库中创建了 House 表。实体类内容如代码清单 4-75 和 4-76 所示。

代码清单 4-75　多数据源项目 City 类代码

```
@Entity
@Table(name="city")
public class City {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY
)
    private int cityId;
    private String cityName;
    private String cityIntroduce;

    //省略 set、get 方法
    ...
}
```

代码清单 4-76　多数据源项目 House 类代码

```
@Entity
@Table(name="house")
public class House {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int houseId;
    private String houseName;
    private String houseIntroduce;

    //省略 set、get 方法
    ...
}
```

接下来分别创建 City 实体和 House 实体对应的数据操作层，内容如代码清单 4-77 和 4-78 所示。

代码清单 4-77　多数据源项目 CityRepositroy 类代码

```
public interface CityRepository extends JpaRepository<City,Integer> {
}
```

代码清单 4-78　多数据源项目 HouseRepository 类代码

```
public interface HouseRepository extends JpaRepository<House,Integer> {
}
```

新建一个 controller，并且在类内创建一个新增的方法，分别对两个数据源进行操作，代码如代码清单 4-79 所示。

代码清单 4-79　多数据源项目 TestController 类代码

```
@RestController
public class TestController {

    @Autowired
    CityRepository cityRepository;

    @Autowired
    HouseRepository houseRepository;

    @GetMapping("/testDataSource")
    public String testDataSource(){
        City city = new City("北京","中国首都");
        cityRepository.save(city);
        House house = new House("豪宅","特别大的豪宅");
        houseRepository.save(house);
        return "success";
    }
}
```

在浏览器上访问 http://localhost:8080/testDataSource，可以看到分别在两个数据库中插入了数据，其他 JPA 操作同样能够进行。感兴趣的读者可以当作一次复习，巩固一下 JPA 的使用。

4.5.4　基于 MyBatis 使用多数据

复制前面的项目，将 JPA 依赖修改为 MyBatis。再使用 MyBatis 对 4.5.3 小节的场景进行复现，对 MyBatis 复习一遍。还是使用 4.5.3 小节的实体类对象，新建 CityMapper 和 HouseMapper，分别利用注解写两个查询方法，即查询所有 City 和 House，代码如代码清单 4-80 和 4-81 所示。

代码清单 4-80　多数据源项目 CityMapper 类代码

```
@Mapper
public interface CityMapper {
    @Select("SELECT * FROM City")
    List<City> getAllCity();
}
```

代码清单 4-81　多数据源项目 HouseMapper 类代码

```
@Mapper
public interface HouseMapper {
    @Select("SELECT * FROM House")
    List<House> getAllHouse();
}
```

使用 MyBatis 操作多数据源与 JPA 的配置略有不同，如代码清单 4-82 所示。

代码清单 4-82　多数据源项目 HouseMapper 类代码

```
@Configuration
@MapperScan(basePackages = "com.springboot.mapper.datasource",
        sqlSessionTemplateRef = "sqlSessionTemplatePrimary")
public class TestDataSourceConfig {
    @Bean(name = "sqlSessionFactoryPrimary")
    @Primary
    public SqlSessionFactory masterSqlSessionFactory(@Qualifier
("testDataSource") DataSource dataSource) throws Exception {
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
        bean.setDataSource(dataSource);
        //如果使用 xml 写 SQL 的话在这里配置
        //bean.setMapperLocations(new PathMatchingResourcePatternResolver().
getResources("classpath:mapper/datasource/*.xml"));
        return bean.getObject();
    }

    @Bean(name = "transactionManagerPrimary")
    @Primary
    public DataSourceTransactionManager masterDataSourceTransactionManager
(@Qualifier("testDataSource") DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean(name = "sqlSessionTemplatePrimary")
    @Primary
```

```
    public SqlSessionTemplate masterSqlSessionTemplate(@Qualifier
("sqlSessionFactoryPrimary") SqlSessionFactory sqlSessionFactory) {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}
```

本文是使用注解的方式执行的 SQL，如果使用 XML 方式写 SQL 的话需要使用 setMapperLocations 方法进行设置。

使用 TestController 进行测试，如代码清单 4-83 所示。

代码清单 4-83　多数据源项目 TestController 代码

```
@RestController
public class TestController {

    @Autowired
    HouseMapper houseMapper;

    @Autowired
    CityMapper cityMapper;

    @GetMapping("/testDataSource")
    public Map testDataSource(){
        Map map = new HashMap();
        List<City> cityList=cityMapper.getAllCity();
        List<House> houseList=houseMapper.getAllHouse();
        map.put("cityList",cityList);
        map.put("houseList",houseList);
        return map;
    }
}
```

使用浏览器访问 http://localhost:8080/testDataSource，可以看到在使用 JPA 时插入的数据。

4.6　使用 Druid 数据库连接池

4.6.1　Druid 简介

Druid 是 Java 语言中最好的数据库连接池，是阿里巴巴的一个开源项目，作为一个优秀的数据库连接池，Druid 提供了优秀的稳定性，并且在性能方面比其他数据库连接提高了很多，最重要的是 Druid 提供了实时监控功能，如数据源监控、SQL 监控、SQL 防火墙、Web 应用监控、URI 监

控、Session 监控、Spring 监控等。正如 Druid 官网（http://druid.io/）介绍的那样：Druid 主要用于存储、查询和分析大型事件流。所谓分析大型事件流，就是前面介绍的监控功能，接下来会一一介绍。

4.6.2　配置 Druid

在 pom 文件中加入 Druid 依赖、MySQL 依赖以及 JPA 依赖，依赖代码如代码 4-84 所示。

代码清单 4-84　Druid 项目依赖代码

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.16</version>
    </dependency>
</dependencies>
```

接下来需要对配置文件进行配置，Druid 的配置有很多，如代码清单 4-85 所示。

代码清单 4-85　多数据源项目配置文件代码

```
##数据库配置
##数据库地址
spring.datasource.url=jdbc:mysql://localhost:3306/test?characterEncoding=
utf8&useSSL=false
##数据库用户名
spring.datasource.username=root
##数据库密码
spring.datasource.password=root
##数据库驱动
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

#这里是不同的
#使用 druid 的话，需要多配置一个属性 spring.datasource.type
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource

# 连接池的配置信息
# 初始化大小，最小或最大
spring.datasource.initialSize=5
spring.datasource.minIdle=5
spring.datasource.maxActive=20
# 配置获取连接等待超时的时间
spring.datasource.maxWait=60000
# 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
spring.datasource.timeBetweenEvictionRunsMillis=60000
# 配置一个连接在池中最小生存的时间，单位是毫秒
spring.datasource.minEvictableIdleTimeMillis=300000
spring.datasource.validationQuery=SELECT 1 FROM DUAL
spring.datasource.testWhileIdle=true
spring.datasource.testOnBorrow=false
spring.datasource.testOnReturn=false
# 打开 PSCache，并且指定每个连接上 PSCache 的大小
spring.datasource.poolPreparedStatements=true
spring.datasource.maxPoolPreparedStatementPerConnectionSize=20
# 配置监控统计拦截的 filters，去掉后监控界面 sql 无法统计，wall 用于防火墙
spring.datasource.filters=stat,wall,log4j
```

与多数据源类似，需要创建一个配置文件接收 Druid 配置，代码如代码清单 4-86 所示。

代码清单 4-86　多数据源项目 DruidConfig 类代码

```
@Configuration
public class DruidConfig {
```

```
@Value("${spring.datasource.url}")
private String dbUrl;

@Value("${spring.datasource.username}")
private String username;

@Value("${spring.datasource.password}")
private String password;

@Value("${spring.datasource.driver-class-name}")
private String driverClassName;

@Value("${spring.datasource.initialSize}")
private int initialSize;

@Value("${spring.datasource.minIdle}")
private int minIdle;

@Value("${spring.datasource.maxActive}")
private int maxActive;

@Value("${spring.datasource.maxWait}")
private int maxWait;

@Value("${spring.datasource.timeBetweenEvictionRunsMillis}")
private int timeBetweenEvictionRunsMillis;

@Value("${spring.datasource.minEvictableIdleTimeMillis}")
private int minEvictableIdleTimeMillis;

@Value("${spring.datasource.validationQuery}")
private String validationQuery;

@Value("${spring.datasource.testWhileIdle}")
private boolean testWhileIdle;

@Value("${spring.datasource.testOnBorrow}")
private boolean testOnBorrow;

@Value("${spring.datasource.testOnReturn}")
private boolean testOnReturn;
```

```
    @Value("${spring.datasource.poolPreparedStatements}")
    private boolean poolPreparedStatements;

    @Value("${spring.datasource. maxPoolPreparedStatementPerConnectionSize}")
    private int maxPoolPreparedStatementPerConnectionSize;

    @Value("${spring.datasource.filters}")
    private String filters;

    @Value("{spring.datasource.connectionProperties}")
    private String connectionProperties;

    @Bean
    @Primary  //主数据源
    public DataSource dataSource(){
        DruidDataSource datasource = new DruidDataSource();

        datasource.setUrl(this.dbUrl);
        datasource.setUsername(username);
        datasource.setPassword(password);
        datasource.setDriverClassName(driverClassName);

        //configuration
        datasource.setInitialSize(initialSize);
        datasource.setMinIdle(minIdle);
        datasource.setMaxActive(maxActive);
        datasource.setMaxWait(maxWait);
        datasource.setTimeBetweenEvictionRunsMillis
(timeBetweenEvictionRunsMillis);
        datasource.setMinEvictableIdleTimeMillis
(minEvictableIdleTimeMillis);
        datasource.setValidationQuery(validationQuery);
        datasource.setTestWhileIdle(testWhileIdle);
        datasource.setTestOnBorrow(testOnBorrow);
        datasource.setTestOnReturn(testOnReturn);
        datasource.setPoolPreparedStatements(poolPreparedStatements);
        datasource.setMaxPoolPreparedStatementPerConnectionSize
(maxPoolPreparedStatementPerConnectionSize);
        try {
            datasource.setFilters(filters);
        } catch (SQLException e) {
        }
```

```
        datasource.setConnectionProperties(connectionProperties);

        return datasource;
    }
}
```

新建一个 Druid 的过滤器，用于过滤一些静态文件，代码如代码清单 4-87 所示。

代码清单 4-87　多数据源项目 TestController 代码

```
@WebFilter(filterName="druidWebStatFilter",urlPatterns="/*",
        initParams={
                @WebInitParam(name="exclusions",value="*.js,*.gif,*.jpg,
*.bmp, *.png,*.css,*.ico,/druid/*")//忽略资源
        }
)
public class DruidFilter extends WebStatFilter  {
}
```

新建 DruidServlet，这个类继承 StatViewServlet，用于过滤管理页面 IP 等信息，代码如代码清单 4-88 所示。

代码清单 4-88　多数据源项目 DruidServlet 类代码

```
@WebServlet(urlPatterns="/druid/*",
        initParams={
                @WebInitParam(name="allow",value=""),
                                    // IP 白名单(若没有配置或者为空，则允许所有访问)
                @WebInitParam(name="deny",value=""),
                                                // IP 黑名单 (deny 优先于 allow)
                @WebInitParam(name="loginUsername",value="admin"),
                                                // 登录 druid 管理页面用户名
                @WebInitParam(name="loginPassword",value="admin")
                                                // 登录 druid 管理页面密码
        })
public class DruidServlet extends StatViewServlet {

}
```

这次需要注意的是，要在启动类上加入@ServletComponentScan，否则无法扫描到 DruidServlet，启动类代码如代码清单 4-89 所示。

代码清单 4-89　多数据源项目 SpringBootDruidApplication 代码

```
@SpringBootApplication
// 启动类必须加入@ServletComponentScan 注解，否则无法扫描到 servlet
@ServletComponentScan
```

```
public class SpringbootDruidApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootDruidApplication.class, args);
    }
}
```

4.6.3 操作数据库

这里以 JPA 操作数据库为例创建一个实体类 User，如代码清单 4-90 所示。

代码清单 4-90 多数据源项目 City 实体类代码

```
@Entity
@Table(name="city")
public class City {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int cityId;
    private String cityName;
    private String cityIntroduce;

    ... //省略 set、get 方法
}
```

接下来分别创建 Repository 和 Controller 进行测试，这里就不具体解释了，代码分别如代码清单 4-91 和代码清单 4-92 所示。

代码清单 4-91 多数据源项目 CityRepository 类代码

```
public interface CityRepository extends JpaRepository<City,Integer> {
}
```

代码清单 4-92 多数据源项目 CityController 类代码

```
@RestController
public class CityController {

    @Autowired
    private CityRepository cityRepository;

    @GetMapping(value = "saveCity")
    public String saveCity(String cityName,String cityIntroduce){
        City city = new City(cityName,cityIntroduce);
        cityRepository.save(city);
```

```
        return "success";
    }

    //http://localhost:8888/deleteCity?cityId=2
    @GetMapping(value = "deleteCity")
    public String deleteCity(int cityId){
        cityRepository.delete(cityId);
        return "success";
    }

    @GetMapping(value = "updateCity")
    public String updateCity(int cityId,String cityName,String cityIntroduce){
        City city = new City(cityId,cityName,cityIntroduce);
        cityRepository.save(city);
        return "success";
    }

    @GetMapping(value = "getCityById")
    public City getCityById(Integer cityId){
        City city = cityRepository.findOne(cityId);
        return city;
    }
}
```

启动项目，访问对应链接测试，这里不再赘述，和之前的一样。

4.6.4　Druid 监控页面介绍

1. 登录页

因为项目结合了 Druid，所以我们可以访问 http://localhost:8080/druid，如图 4-18 所示。

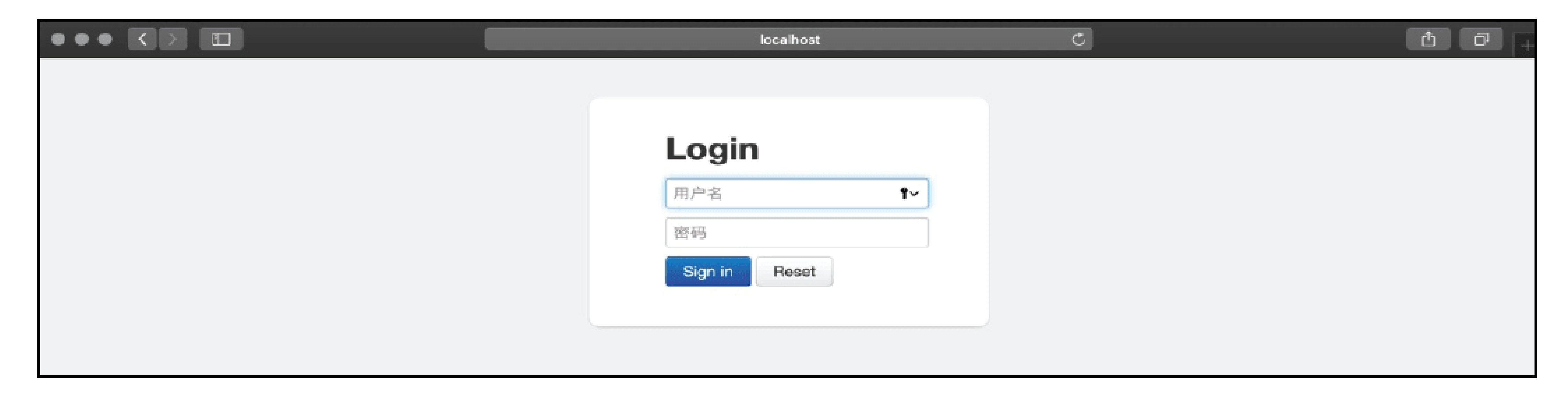


图 4-18　Druid 项目 Druid 监控登录页面

2. 首页

使用用户名 admin、密码 admin 登录后，可以看到如图 4-19 所示的页面。

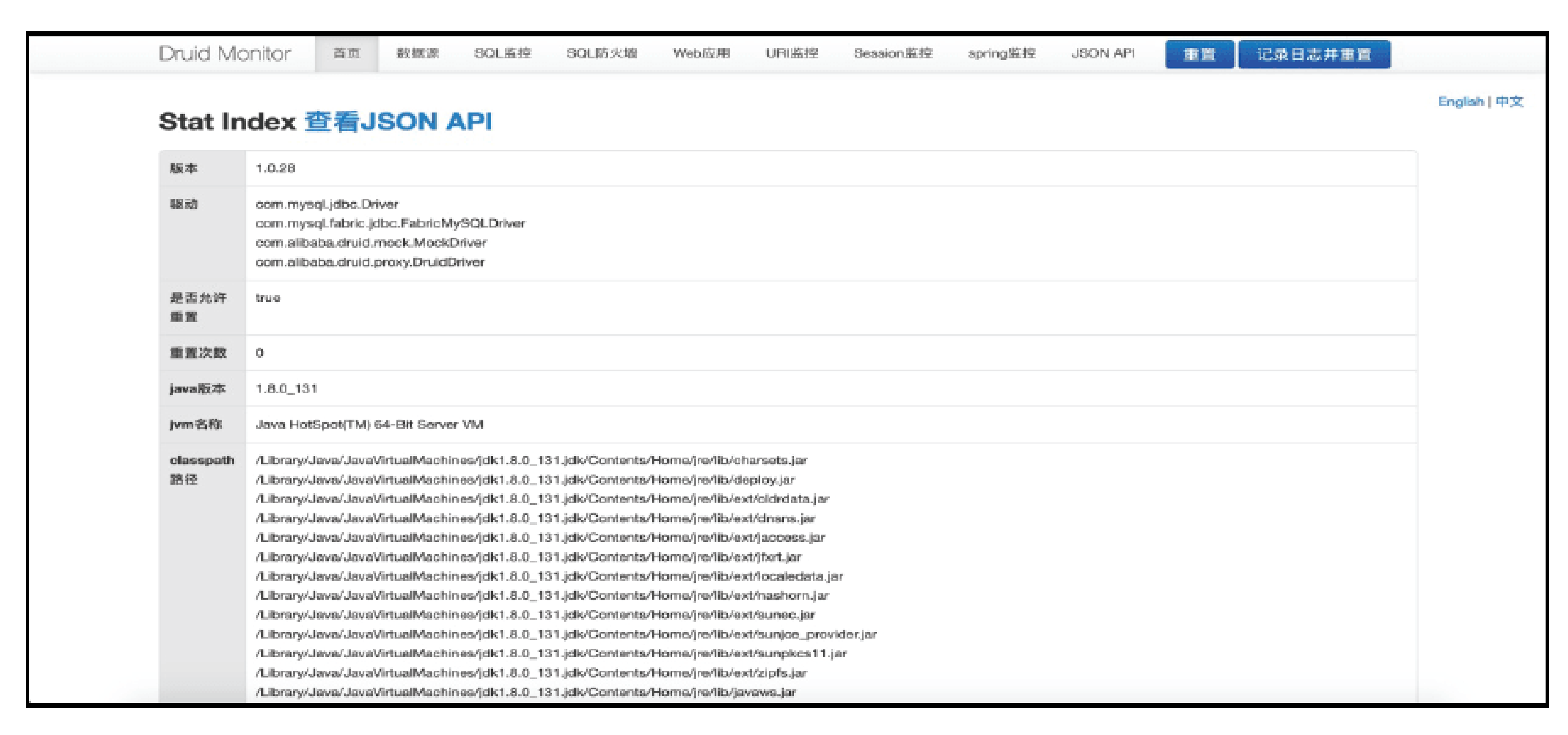


图 4-19　Druid 项目 Druid 首页

从图 4-19 中可以看到，首页包含如下信息。

- 版本：Druid 版本。
- 驱动：包含 Druid 驱动、MySQL 驱动等。
- 是否允许重置。
- 重置次数。
- Java 版本。
- jvm 名称。
- classpath 路径：这里是本机配置的环境变量，由于笔者的电脑配置了太多环境变量，因此导致这一张图没有展示完全。
- 启动时间：项目的启动时间。

3. 数据源页面

接下来，在菜单栏单击数据源菜单，会展示很多配置，基本上都是与数据源配置信息相关的，如果含有多个数据源，就会显示多个 Tab 页。Druid 更加友好的是在最后一列对当前行进行了介绍，具体如图 4-20~图 4-23 所示。

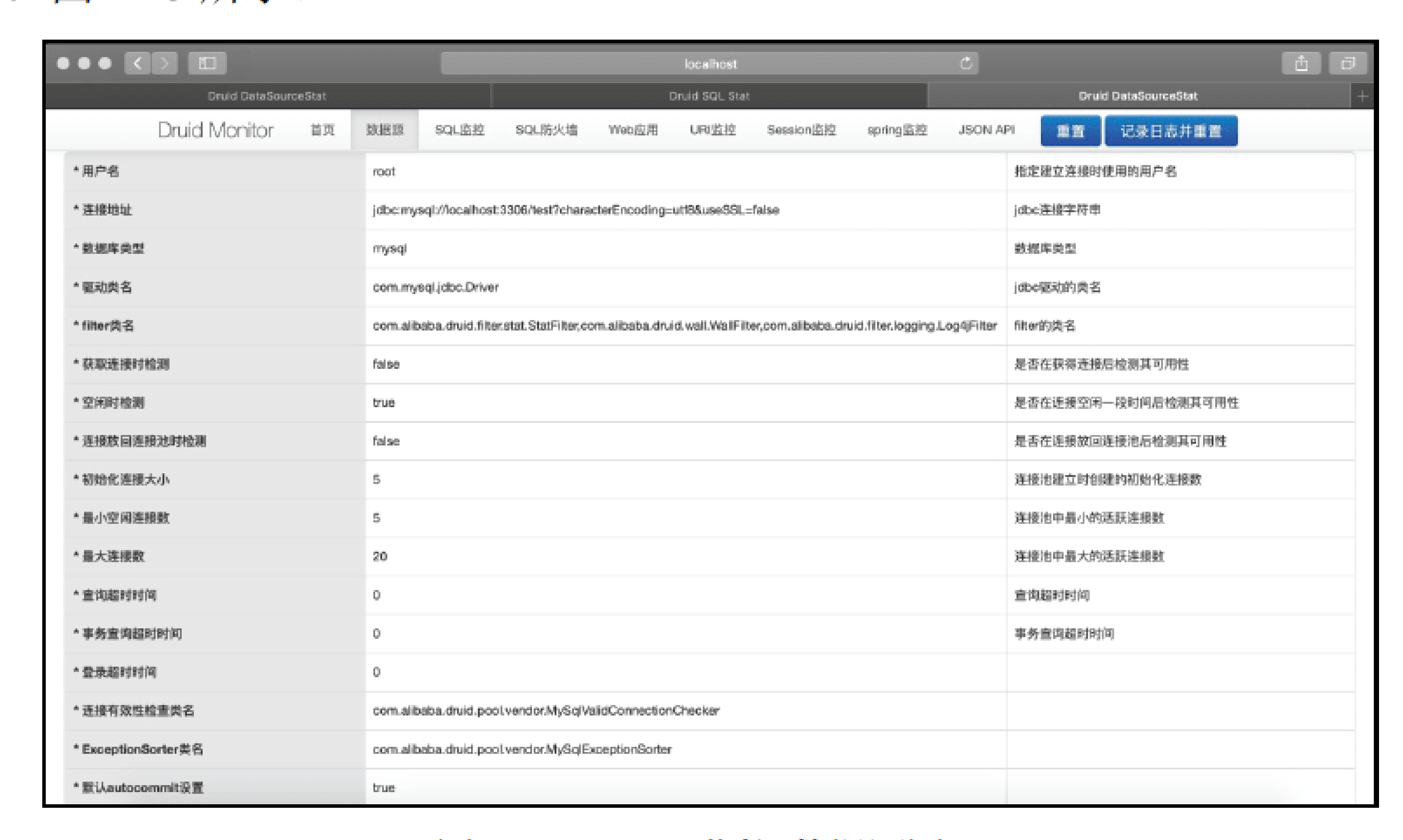


图 4-20　Druid 监控-数据源图一

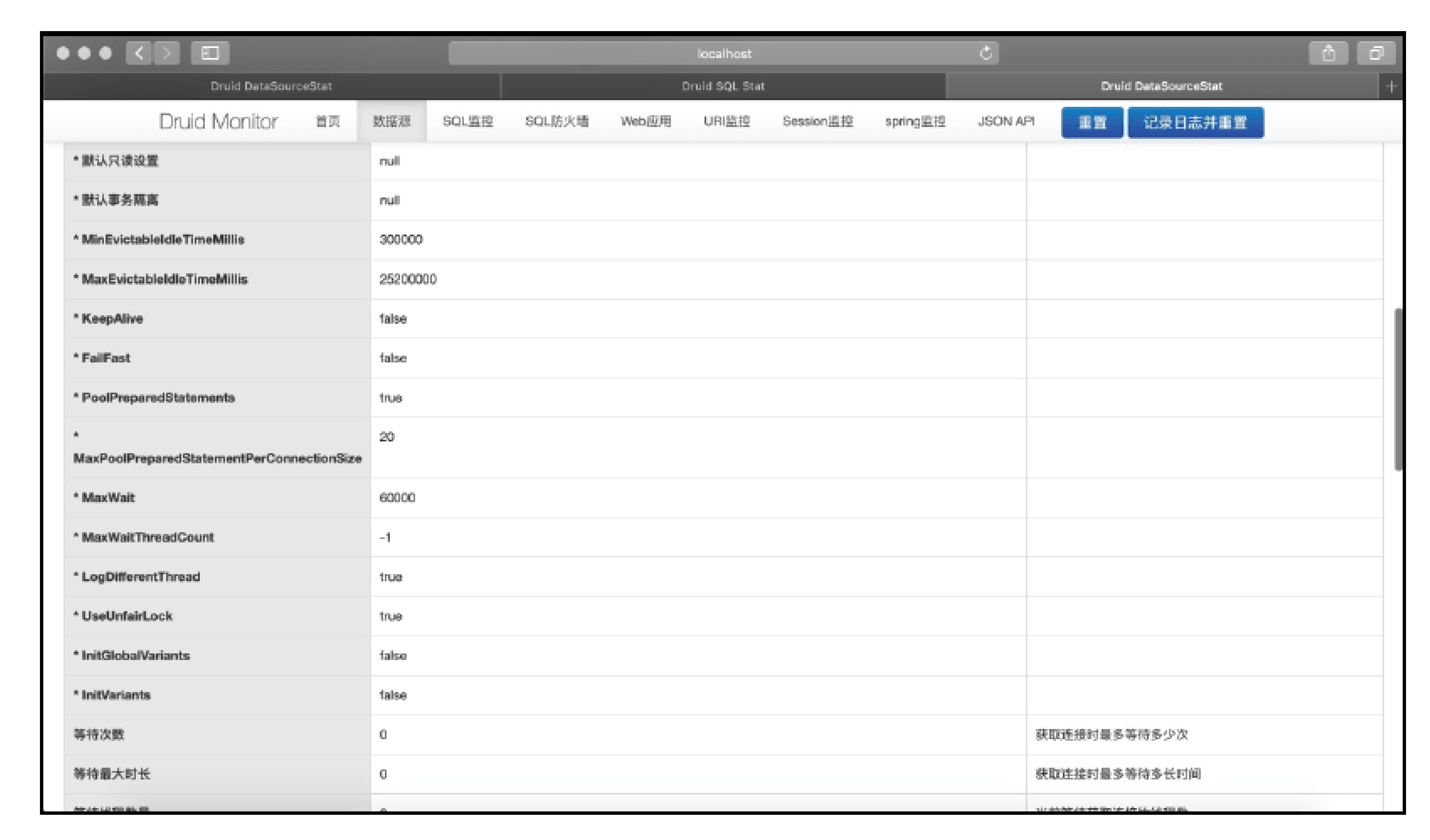


图 4-21　Druid 监控-数据源图二

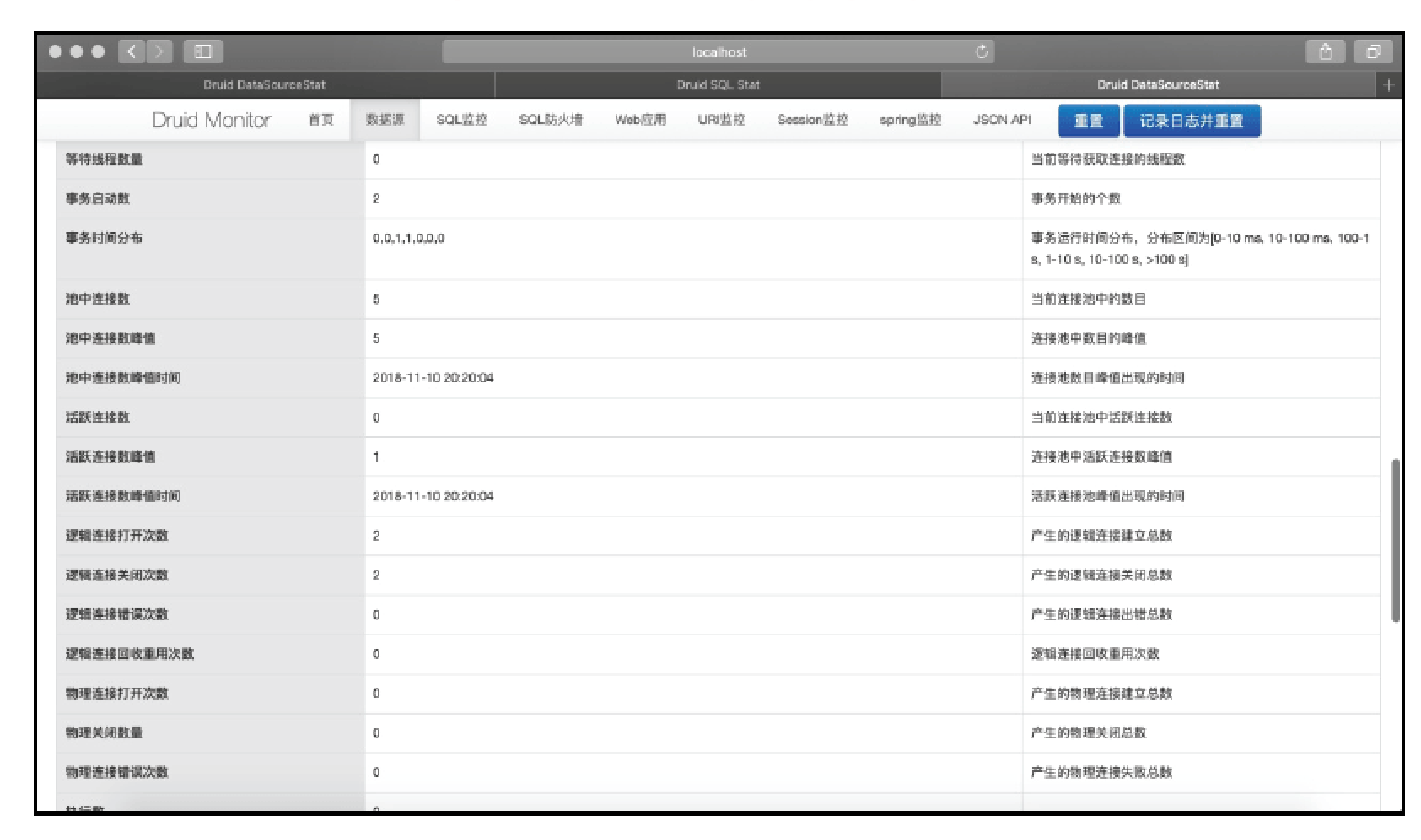


图 4-22　Druid 监控-数据源图三

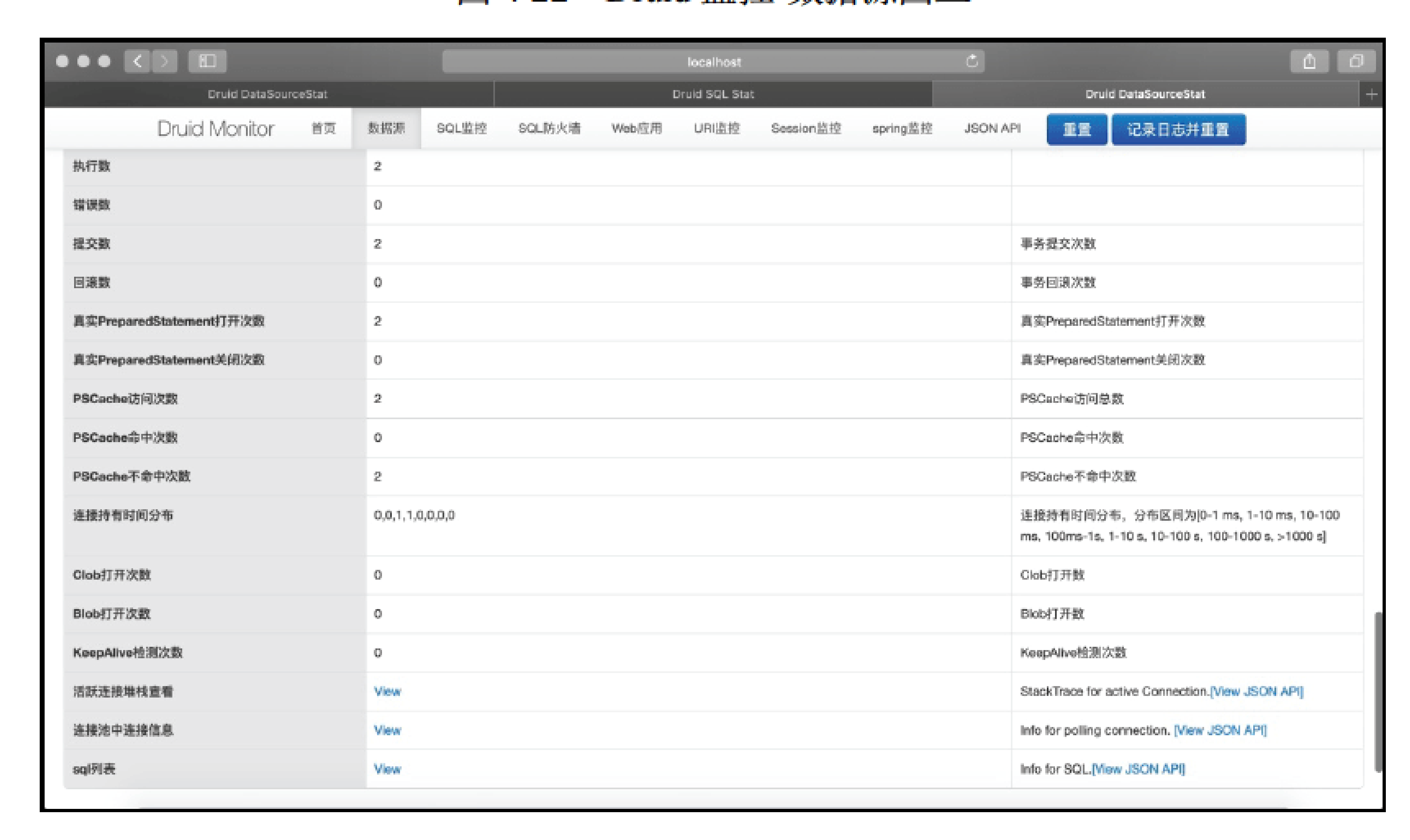


图 4-23　Druid 监控-数据源图四

4. SQL 监控页面

单击菜单栏的 SQL 监控菜单，进入 SQL 监控页面，一开始是没有数据的，这时我们请求两次服务，然后刷新页面，可以看到页面如图 4-24 所示。

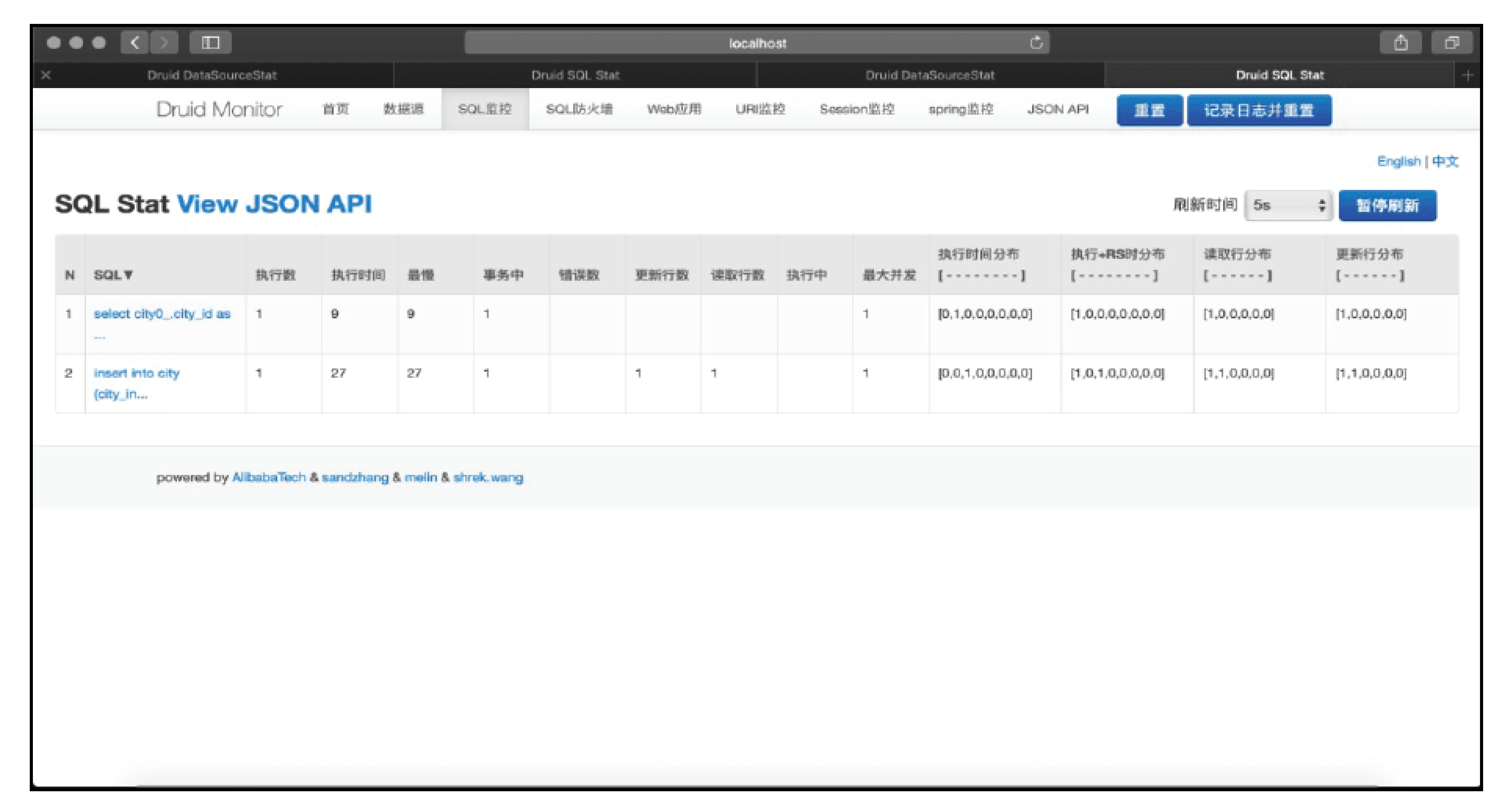


图 4-24 Druid 监控-SQL 监控页面

在图中显示了每条 SQL 的执行次数、时间、事务、最慢 SQL、最大并发、读取行数等信息，并且可以根据每一列的条件进行正序或者倒叙排序，这很有利于我们对系统 SQL 进行问题排查及优化。

5. SQL 防火墙页面

这个页面有针对数据库的保护策略，可以看到表访问统计、白名单、黑名单等数据访问情况，如图 4-25 和图 4-26 所示。

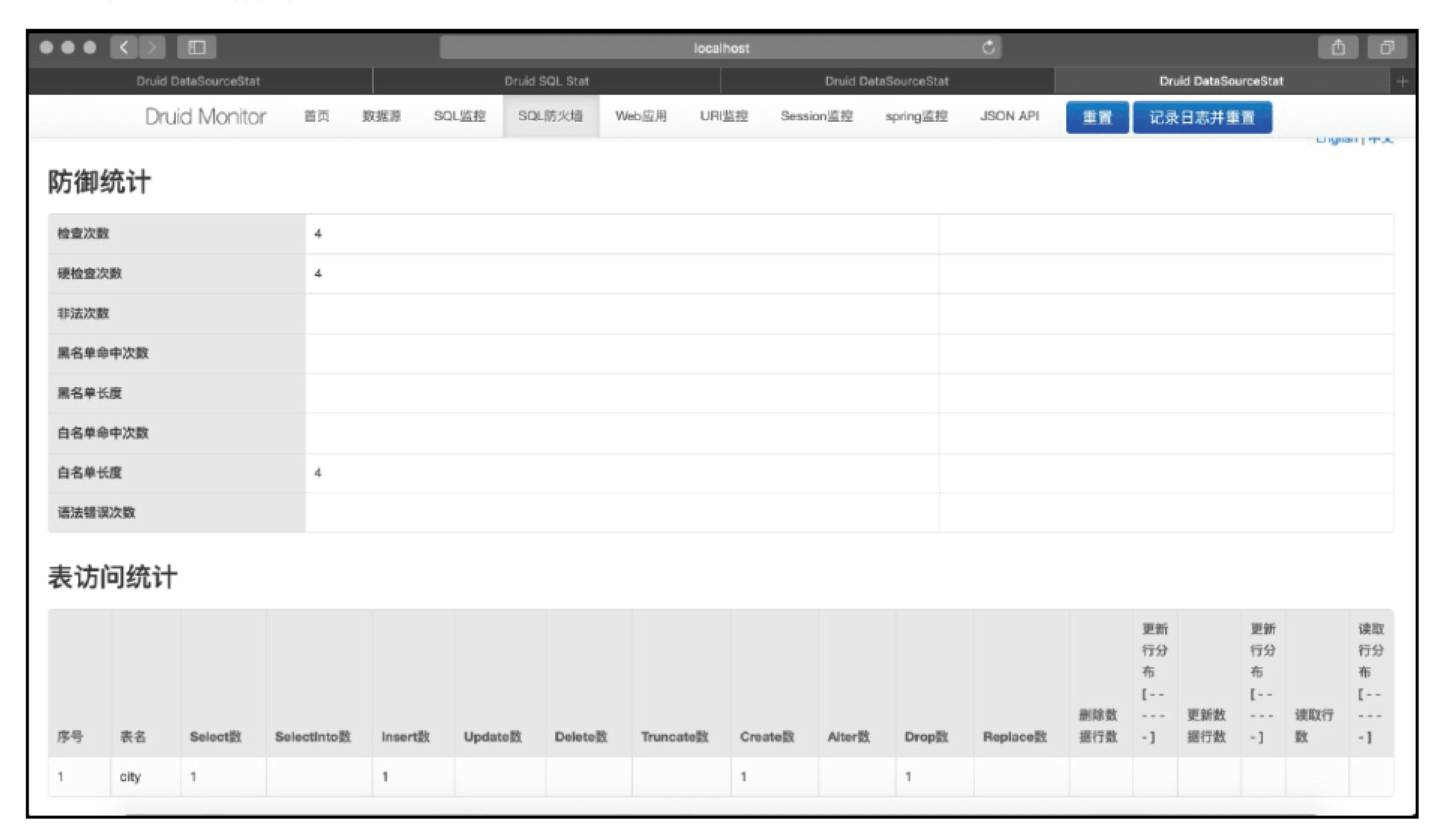


图 4-25 Druid 监控-SQL 防火墙页面图一

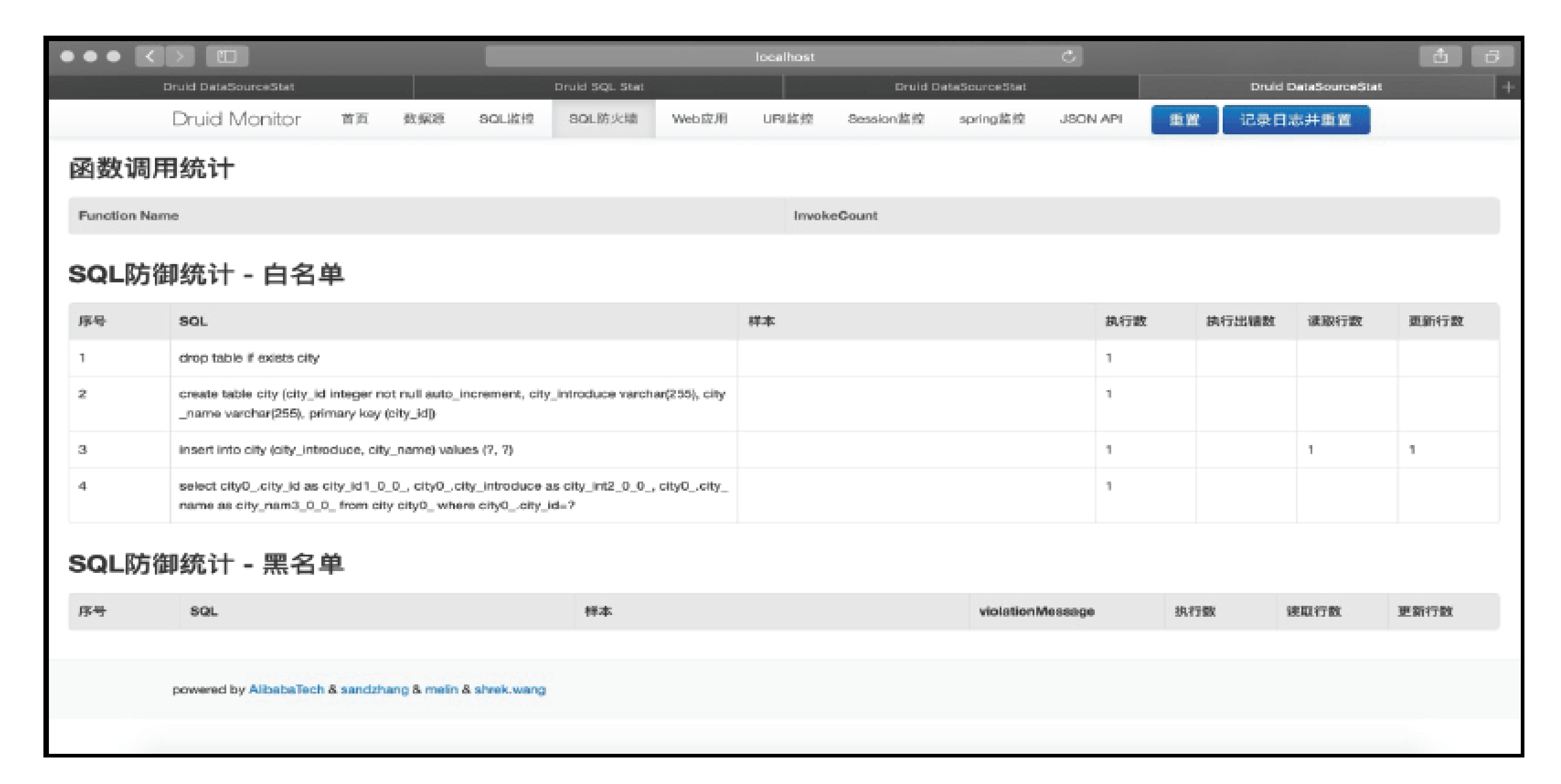


图 4-26　Druid 监控-SQL 防火墙页面图二

6. Web 应用页面

Web 应用页面主要用于实时分析数据库的访问情况，如正在执行多少 SQL、最大并发、事务等，这里只截取了一张图片供参考，感兴趣的读者可以自行研究，如图 4-27 所示。

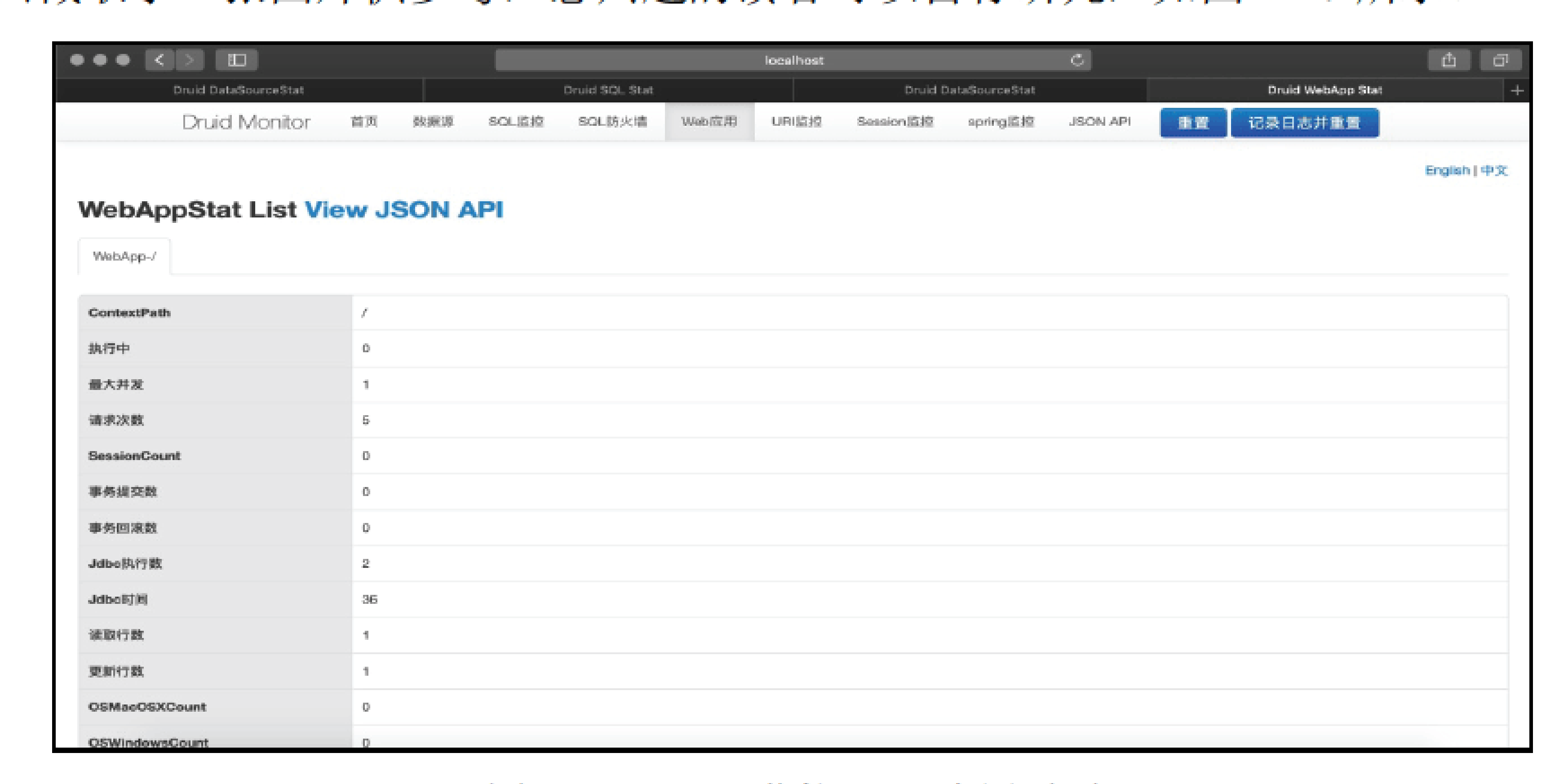


图 4-27　Druid 监控-Web 应用页面

7. URI 监控页面

URI 监控页面主要用于监控对系统的请求，内容大致和前几个监控差不多，有事务、执行数、并发等，如图 4-28 所示。

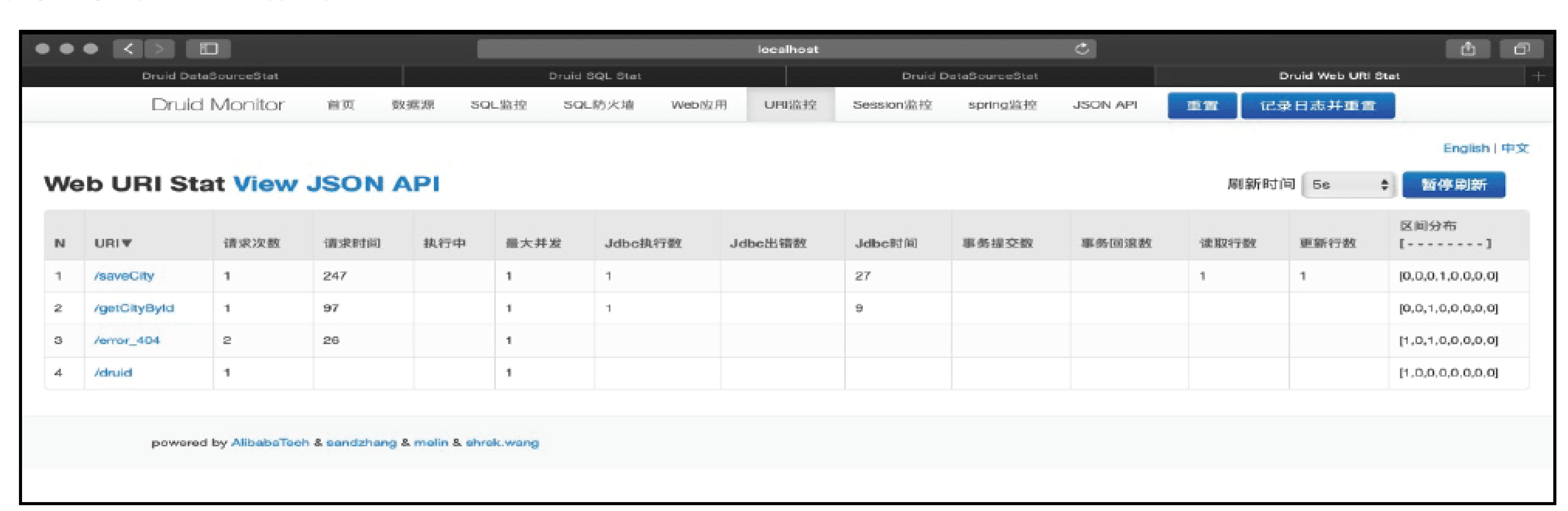


图 4-28　Druid 监控-URI 监控页面

8. Session 监控页面

Session 监控页面主要用于监控系统内 Session 的使用情况，由于本项目没有使用，因此这里没有数据，如图 4-29 所示。

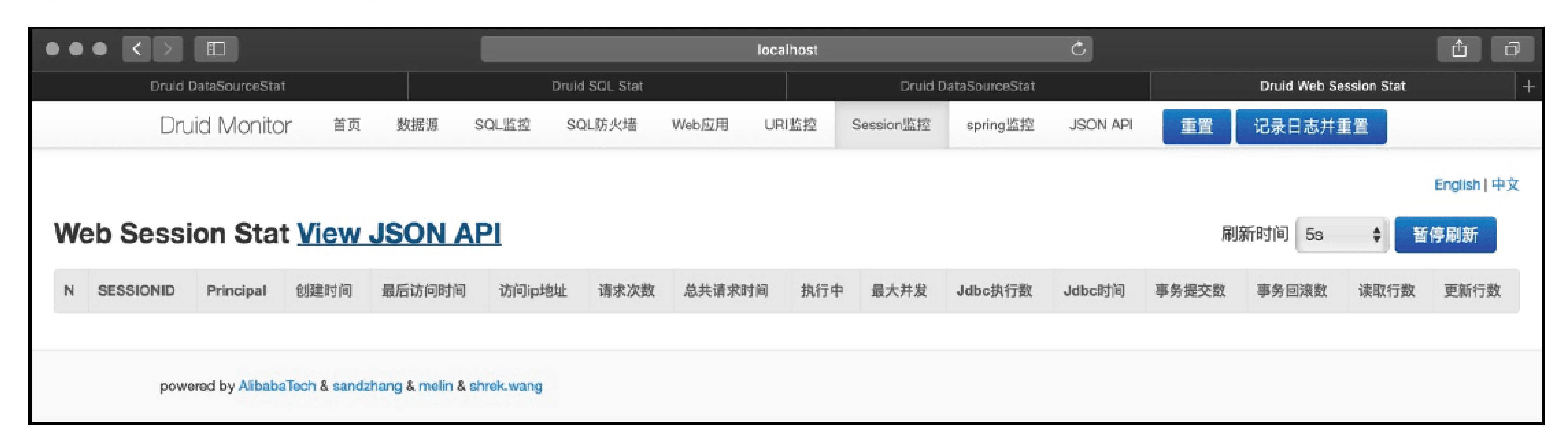


图 4-29　Druid 监控-Session 监控页面-无数据

我们修改一下 getCityById 方法，将查询出来的 city 存入 Session，这里的使用可能不恰当，仅仅是为了查看 Session 监控页面的数据，如代码清单 4-93 所示。

代码清单 4-93　Druid 项目 getCityById 方法改造

```
@GetMapping(value = "getCityById")
public City getCityById(Integer cityId, HttpServletRequest request){
    Optional<City> optionalCity = cityRepository.findById(cityId);
    HttpSession session = request.getSession();
    City city = optionalCity.isPresent() ? optionalCity.get() : null;
    session.setAttribute(cityId.toString(),city);
    return city;
}
```

然后查看 Session，如图 4-30 所示，可以看到一条 Session 数据。

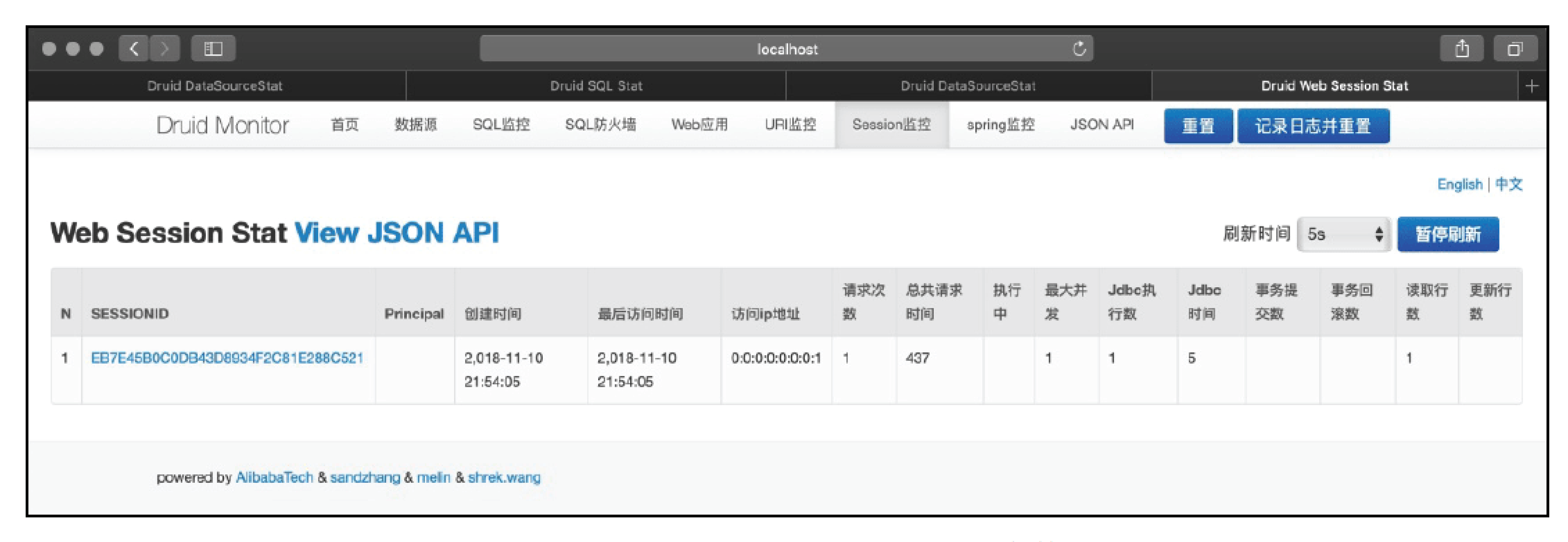


图 4-30　Druid 监控-Session 监控页面-有数据

9. Spring 监控页面

Spring 监控页面主要用于监控系统内 Spring 的使用情况，由于本项目没有使用，因此这里没有数据，如图 4-31 所示。



图 4-31　Druid 监控-Spring 监控页面

10. JSON API

JSON API 页面其实就是对 Druid 监控的一些说明，如图 4-32 所示。

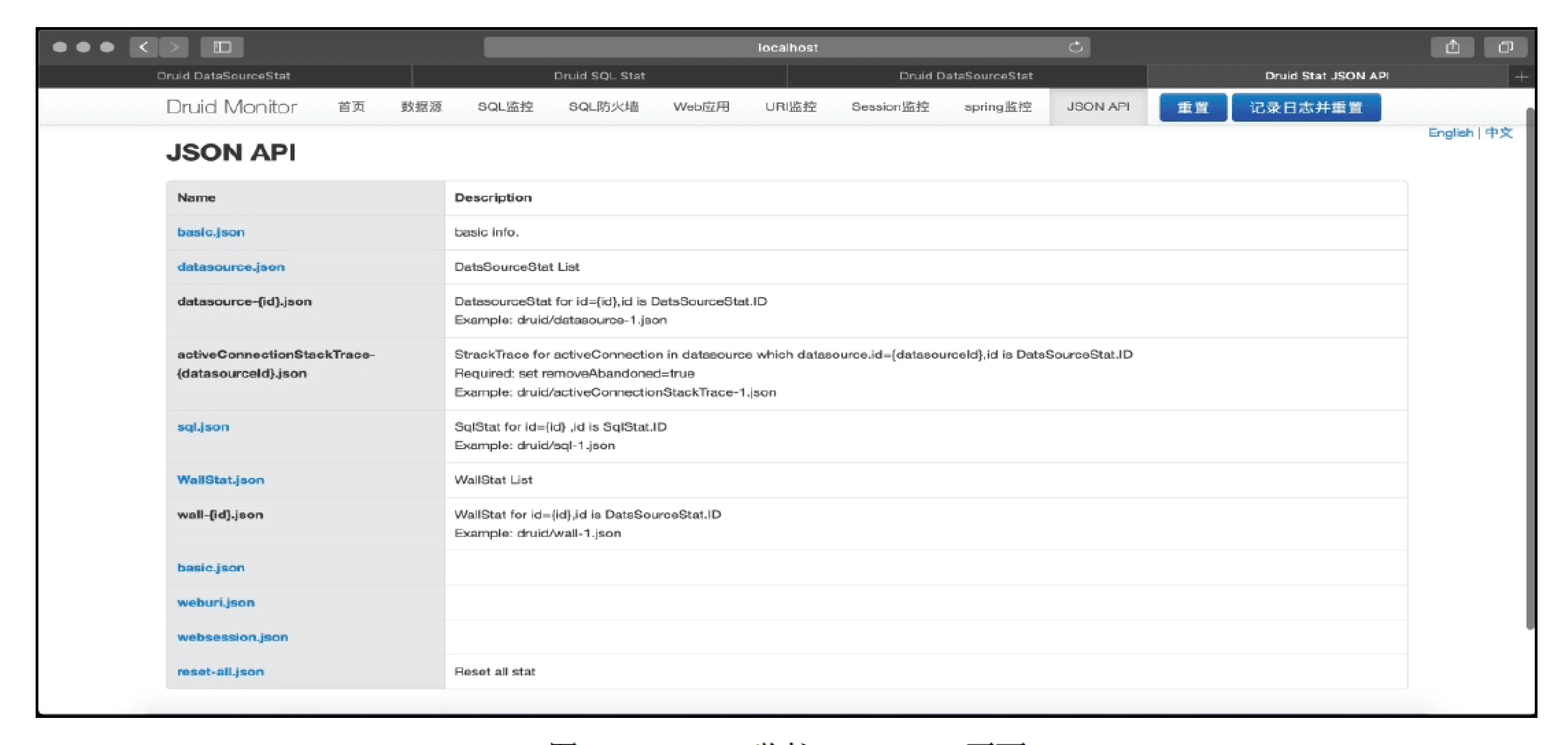


图 4-32　Druid 监控-JSON API 页面

4.7　小　　结

本章对 Spring Boot 使用数据库进行了全面的介绍，从简单的数据库配置到现在特别流行的 JPA 和 MyBatis 的使用都进行了详尽的介绍，相信你已经对操作数据库有了一定的认识，并且熟练运用，可以独立进行开发了。

第 5 章

Spring Boot 的缓存之旅

第 4 章我们学习了数据库的使用，但是数据库并不能完全高性能地解决任何事情，这个时候缓存就出现了。缓存这个词对于很多人来说可能并不陌生，无论是从事传统项目的开发者，还是互联网项目的开发者，可能都对缓存有一定的了解。缓存数据交换的缓冲区，一般来说会将访问量比较大的数据从数据库中查询出来放入缓存中，当下次获取数据的时候，直接从缓存中获取。通常缓存会放入内存或硬盘中，方便开发者使用。本章将对 Spring Boot 如何使用缓存进行学习。

5.1 使用 Spring Cache

5.1.1 Spring Cache 简介

Spring Cache 是 Spring 3.1 以后引入的新技术。它并不像正常缓存那样存储数据，其核心思想是这样的：当我们在调用一个缓存方法时，会把该方法参数和返回结果作为一个键值对存放在缓存中，等到下次利用同样的参数来调用该方法时将不再执行该方法，而是直接从缓存中获取结果进行返回，从而实现缓存的功能。Spring Cache 的使用和 Spring 对于事务管理的使用类似，可以基于注解使用或者基于 XML 配置方式使用。下面我们来学习基于注解的使用。

在 Spring 中提供了 3 个注解来使用 Spring Cache，下面分别进行介绍。

1. @Cacheable

@Cacheable 注解用于标记缓存，也就是对使用@Cacheable 注解的位置进行缓存。@Cacheable 可以在方法或者类上进行标记，当对方法进行标记时，表示此方法支持缓存；当对类进行标记时，表明当前类中所有的方法都支持缓存。在支持 Spring Cache 的环境下，对于使用@Cacheable 标记的方法，Spring 在每次调用方法前都会根据 key 查询当前 Cache 中是否存在相同 key 的缓存元素，

如果存在，就不再执行该方法，而是直接从缓存中获取结果进行返回，否则执行该方法并将返回结果存入指定的缓存中。在使用@Cacheable 时通常会搭配 3 个属性进行使用，分别介绍如下。

- value：在使用@Cacheable 注解的时候，value 属性是必须要指定的，这个属性用于指定 Cache 的名称，也就是说，表明当前缓存的返回值用于哪个缓存上。
- key：和名称一样，用于指定缓存对应的 key。key 属性不是必须指定的，如果我们没有指定 key，Spring 就会为我们使用默认策略生成的 key。其中默认策略是这样规定的：如果当前缓存方法没有参数，那么当前 key 为 0；如果当前缓存方法有一个参数，那么以 key 为参数值；如果当前缓存方法有多个参数，那么 key 为所有参数的 hashcode 值。当然，我们也可以通过 Spring 提供的 EL 表达式来指定当前缓存方法的 key。通常来说，我们可以使用当前缓存方法的参数指定 key，一般为“#参数名”。如果参数为对象，就可以使用对象的属性指定 key，比如使用之前的 User 类，使用方式如代码清单 5-1 所示。

代码清单 5-1　使用类属性作为@Cacheable 注解 key 属性

```
@Cacheable(value="users", key="#user.id")
public User findUser(User user) {
   return new User();
}
```

同时，Spring 框架还为我们提供了 root 对象来使用 key 属性，在指定 key 属性时可以忽略#root，因为 Spring 默认调用的就是 root 对象的属性。其中 root 对象分别内置如下几个属性。

（1）methodName：当前方法的名称，key 值为#root.methodName 或 methodName。
（2）method：指定当前方法，key 值为#root.method.name 或 method.name。
（3）target：当前被调用的对象，key 值为#root.target 或 target。
（4）targetClass：当前被调用的对象的 class，key 值为#root.targetClass 或 targetClass。
（5）args：当前方法参数组成的数组，key 值为#root.args[0]或 args[0] 。
（6）caches：当前被调用的方法使用的 Cache，key 值为#root.caches[0].name 或 caches[0].name。

- condition：主要用于指定当前缓存的触发条件。很多情况下可能并不需要使用所有缓存的方法进行缓存，所以 Spring Cache 为我们提供了这种属性来排除一些特定的情况。以属性指定 key 为 user.id 为例，比如我们只需要 id 为偶数才进行缓存，进行配置 condition 属性的过程如代码清单 5-2 所示。

代码清单 5-2　使用类属性作为@Cacheable 注解 condition 属性

```
@Cacheable(value="users", key="#user.id" , condition="#user.id%2==0")
public User findUser(User user) {
   return new User();
}
```

2. @CachePut

从名称上来看，@CachePut 只是用于将标记该注解的方法的返回值放入缓存中，无论缓存中是否包含当前缓存，只是以键值的形式将执行结果放入缓存中。在使用方面，@CachePut 注解和@Cacheable 注解一致，这里就不具体介绍了。

3. @CacheEvict

Spring Cache 提供了@CacheEvict 注解用于清除缓存数据，与@Cacheable 类似，不过@CacheEvict 用于方法时清除当前方法的缓存，用于类时清除当前类所有方法的缓存。@CacheEvict 除了提供与@Cacheable 一致的 3 个属性外，还提供了一个常用的属性 allEntries，这个属性的默认值为 false，如果指定属性值为 true，就会清除当前 value 值的所有缓存。

5.1.2 配置 Spring Cache 依赖

创建一个项目，在项目中加入 spring-boot-starter-cache 依赖，因为配合数据库操作缓存才会更加明显，所以加入 JPA 和 MySQL 进行测试，依赖代码如代码清单 5-3 所示。

代码清单 5-3 Spring Cache 项目依赖文件代码

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-cache</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

5.1.3 测试运行

配置文件内容与使用 JPA 操作数据库的配置一样，实体类可以复制第 4 章使用的 User 实体类，数据库操作层只是简单地继承了 JpaRepository，代码就不在这里展示了。需要注意的是，我们需要在启动类上加入@EnableCaching 注解，表明启动缓存，启动类代码如代码清单 5-4 所示。

代码清单 5-4　Spring Cache 项目启动类代码

```
@SpringBootApplication
//开启缓存
@EnableCaching
public class SpringbootCacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringbootCacheApplication.class, args);
    }
}
```

还是一如既往地使用 Controller 进行测试。首先看一段代码，然后对代码进行解释，示例代码比较简单，并没有使用 Service 编写代码，正常开发过程中是需要 Service 和 Impl 类对代码进行规范的。示例代码如代码清单 5-5 所示。

代码清单 5-5　Spring Cache 项目 UserController 类代码

```
@RestController
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/saveUser")
    @CachePut(value = "user", key = "#id")
    public User saveUser(Long id, String userName, String userPassword){
        User user = new User(id,userName, userPassword);
        userRepository.save(user);
        return user;
    }

    @GetMapping("/queryUser")
    @Cacheable(value = "user", key = "#id")
    public Optional<User> queryUser(Long id){
        return userRepository.findById(id);
    }

    @GetMapping("/deleteUser")
```

```
    @CacheEvict(value = "user", key = "#id")
    public String deleteUser(Long id){
        userRepository.deleteById(id);
        return "success";
    }

    @GetMapping("/deleteCache")
    @CacheEvict(value = "user", allEntries = true)
    public void deleteCache() {
    }
}
```

对代码清单 5-5 进行分析，其中包含 4 个方法、3 个注解，分别说明如下。

（1）saveUser 方法：其中使用到@CachePut，将返回值存放于缓存中。

（2）queryUser 方法：其中使用到@Cacheable，这个注解在执行前会查看是否已经存在缓存，如果存在，就直接返回；如果不存在，就将返回值存入缓存后再返回。

（3）deleteUser 方法：其中使用到@CacheEvict，用于删除对应的缓存。

（4）deleteCache 方法：与 deleteUser 方法类似，不过这个例子配置了 allEntries 属性，用于删除所有 value 为 user 的缓存。

5.1.4 验证缓存

验证一：@CachePut 注解存储缓存

介绍完注解，接着就要验证了。先在浏览器上访问 http://localhost:8080/saveUser?id=1&userName=dalaoyang&userPassword=123，保存一个 id 为 1 的用户，刚刚介绍了，这个注解会将返回值存入缓存，@CachePut 注解验证完成。

验证二：@Cacheable 注解存储缓存和查询缓存

接下来，我们清空控制台数据，因为在配置文件中设置了打印 SQL，如果查询数据库，就会在控制台打印 SQL；如果使用缓存，就会没有显示。接下来在浏览器上访问 http://localhost:8080/queryUser?id=1，观看控制台发现没有任何日志，证明缓存生效了。

接下来我们在数据库中插入一条 ID 为 2 的 User 数据。继续验证@Cacheable 注解，第一次访问 http://localhost:8080/queryUser?id=2，控制台打印了 SQL，第二次访问没有打印，证明缓存生效，@Cacheable 验证完成。

验证三：@CacheEvict 注解清除缓存

接下来验证@CacheEvict 注解，我们调用 http://localhost:8080/deleteUser?id=1 删除 ID 为 1 的用户，再查询它，发现打印了 SQL，证明缓存已经删除。最后调用 http://localhost:8080/deleteCache 删除全部缓存，再访问 http://localhost:8080/queryUser?id=2，发现打印了 SQL，@CacheEvict 注解验证完成。

由于验证比较简单，因此只给出了文字验证说明，感兴趣的读者可以自己结合更多场景进行验证。

5.2　使用 Redis

5.2.1　Redis 简介

Redis 是一个高性能的缓存存储系统，并且以 Key-Value 的形式存储数据。目前 Value 支持 5 种数据类型，其中包括 string（字符串）、list（链表）、set（集合）、zset（sorted set，有序集合）和 hash（哈希类型）。Redis 支持多种开发语言，如 Java、C/C++、C#、PHP、JavaScript、Perl、Objective-C、Python、Ruby、Erlang 等。同时，Redis 还支持数据的持久化，不只可以将数据存储在内存中，还可以将数据存储到硬盘内，不需要担心数据的丢失。在性能方面，Redis 官方（官网地址：https://redis.io/）提供了这样的数据：读的速度是 110 000 次/s，写的速度是 81 000 次/s，是一个真正的高性能数据库。

5.1 节简单介绍了有关 Spring Boot 使用 Redis 方面相关的依赖及配置，本节将具体学习 Spring Boot 对于 Redis 的使用。

5.2.2　项目配置

在创建项目之前，需要启动 Redis。启动 Redis 后，新建项目，在 pom 文件中加入 Redis 依赖，依赖代码如代码清单 5-6 所示。

代码清单 5-6　Spring Boot 使用 Redis 项目依赖文件代码

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

在配置文件中配置 Redis 信息，这里我们只配置 Redis 服务地址和端口，如代码清单 5-7 所示。

代码清单 5-7　Spring Boot 使用 Redis 项目依赖文件代码

```
# Redis 服务器地址
spring.redis.host=localhost
# Redis 服务器连接端口
spring.redis.port=6379
```

对于使用 Redis，常用操作无非就是 set 方法和 get 方法，所以我们创建一个 RedisService 对这两个操作进行提取，在类上加入@Service，表明这是一个受 Spring 管理的 JavaBean 对象，在 RedisService 类内注入 RedisTemplate，用于对 Redis 缓存进行操作。创建两个方法：Set 方法和 Get 方法，分别用于使用 RedisTemplate 进行存放数据和取出数据，如代码清单 5-8 所示。

代码清单 5-8　Spring Boot 使用 Redis 项目 RedisService 类代码

```
@Service
public class RedisService {
    @Resource
    private RedisTemplate<String,Object> redisTemplate;

    public void set(String key, Object value) {
        ValueOperations<String,Object> vo = redisTemplate.opsForValue();
        vo.set(key, value);
    }

    public Object get(String key) {
        ValueOperations<String,Object> vo = redisTemplate.opsForValue();
        return vo.get(key);
    }
}
```

还是使用之前的 User 实体类。接下来创建一个 UserController 进行测试，分别调用 RedisService 内的 Set 方法和 Get 方法，内容比较简单，直接看代码即可，如代码清单 5-9 所示。

代码清单 5-9　Spring Boot 使用 Redis 项目 UserController 类代码

```
@RestController
public class UserController {
    @Autowired
    private RedisService redisService;

    @GetMapping(value = "saveUser")
    public String saveUser(Long id,String userName,String userPassword){
        User user = new User(id,userName,userPassword);
        redisService.set(id.toString(),user);
```

```
        return "success";
    }

    @GetMapping(value = "getUserById")
    public Object getUserById(Long id){
        return redisService.get(id.toString());
    }
}
```

5.2.3　测试运行

启动项目，访问 http://localhost:8080/saveUser?id=1&userName=dalaoyang&userPassword=123，查看控制台发现报错了。

根据控制台提示分析，原因是我们没有对实体类进行序列化，对实体类进行序列化后重启项目，再次访问发现可以保存了。接下来查看一下 Redis 数据库，发现值虽然存进来了，但是编码格式似乎不对，如图所 5-1 所示。

```
redis-4.0.8 — redis-cli — 80×24
127.0.0.1:6379> keys *
1) "\xac\xed\x00\x05t\x00\x011"
127.0.0.1:6379>
```

图 5-1　redis-cli 查看图一

从图 5-1 中可以看出，Redis 存入的 key 编码有一些问题。接下来我们修改一下 set 方法，将 key 值进行序列化，修改 set 方法后的代码如代码清单 5-10 所示。

代码清单 5-10　Spring Boot 使用 Redis 项目 set 方法修改后代码

```
public void set(String key, Object value) {
    //在 redis 里面查看 key 编码问题
    redisTemplate.setKeySerializer(new StringRedisSerializer());
    ValueOperations<String,Object> vo = redisTemplate.opsForValue();
    vo.set(key, value);
}
```

重新启动项目，为了保证效果清晰，先使用 flushall（这个指令用于清空 Redis 的所有 key）清空 key，再调用刚刚保存的方法，这时使用 redis-cli 查看刚刚保存的 key，如图 5-2 所示。

```
redis-4.0.8 — redis-cli — 80×24
127.0.0.1:6379> keys *
1) "1"
127.0.0.1:6379>
```

图 5-2　redis-cli 查看图二

接下来在 redis-cli 中查看 key 为 1 的值，使用命令 get 1 获取 key 为 1 的数据，如图 5-3 所示。

```
127.0.0.1:6379> get 1
"\xac\xed\x00\x05sr\x00\x19com.dalaoyang.entity.User\xc1W\\*\x89 \x90v\x02\x00\x
03L\x00\x02idt\x00\x10Ljava/lang/Long;L\x00\buserNamet\x00\x12Ljava/lang/String;
L\x00\x0cuserPasswordq\x00~\x00\x02xpsr\x00\x0ejava.lang.Long;\x8b\xe4\x90\xcc\x
8f#\xdf\x02\x00\x01J\x00\x05valuexr\x00\x10java.lang.Number\x86\xac\x95\x1d\x0b\
x94\xe0\x8b\x02\x00\x00xp\x00\x00\x00\x00\x00\x00\x00\x01t\x00\tdalaoyangt\x00\x
03123"
127.0.0.1:6379>
```

图 5-3 redis-cli 查看图三

从图 5-3 中可以看出，value 的编码似乎也有一些问题。我们继续改造 set 方法，使用 Jackson2JsonRedisSerializer 对 User 类数据进行实例化编码，修改 set 方法后如代码清单 5-11 所示。

代码清单 5-11 Spring Boot 使用 Redis 项目修改 set 方法后的代码

```
    public void set(String key, Object value) {
            redisTemplate.setKeySerializer(new StringRedisSerializer());
            redisTemplate.setValueSerializer(new 
Jackson2JsonRedisSerializer(Object.class));
            ValueOperations<String,Object> vo = redisTemplate.opsForValue();
            vo.set(key, value);
    }
```

重新启动项目后，再次插入数据，这时就不需要对 Redis 进行清空了，因为相同 key 的数据会覆盖，如图 5-4 所示。

```
127.0.0.1:6379> get 1
"{\"id\":1,\"userName\":\"dalaoyang\",\"userPassword\":\"123\"}"
127.0.0.1:6379>
```

图 5-4 redis-cli 查看图四

接下来用 get 方法对刚刚 set 方法存储的值进行取值，在浏览器上访问 http://localhost:8080/getUserById?id=1，可以看到浏览器显示了如下数据：

```
{"id":1,"userName":"dalaoyang","userPassword":"123"}
```

到这里，set 方法已经测试完毕。由于样例代码只用了这两种数据结构，因此只对这两种结构进行实例化。其实还提供了很多种类型的实例化模板供我们使用，具体可以根据需要修改。

5.2.4 使用 Redis 缓存

在刚刚对 Redis 的使用中，只是使用 Redis 作为数据库。接下来我们利用 Redis 作为缓存数据库进而分担数据库的压力。其实在真实的开发环境中，一般都是以传统数据库为主、缓存数据库为辅，这样就可以极大地提高数据库的访问性能。

接下来我们继续改造项目，其实大致原理是这样的，首先向数据库中插入一条数据，然后查询这条数据，在第一次查询的时候从数据库查询，然后将数据存入 Redis 并设置过期时间，其间，再次查询就会先查询 Redis，如果在 Redis 内没有查到，重复这个流程。以 JPA 操作数据库为例，加入 JPA 依赖以及 MySQL 依赖、配置文件及 UserRepository，引用 5.2.3 节的内容即可。依赖内容如代码清单 5-12 所示。

代码清单 5-12　Spring Boot 使用 Redis 项目引入 JPA 依赖和 MySQL 依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

接下来，在 RedisService 内新增一个可以设置过期时间的方法，其实就是在 set 方法中新增两个参数：过期时间和过期时间单位。新增 set 方法如代码清单 5-13 所示。

代码清单 5-13　Spring Boot 使用 Redis 项目 RedisService 类新增 set 方法代码

```
    public void set(String key, Object value , Long time, TimeUnit t) {
            redisTemplate.setKeySerializer(new StringRedisSerializer());
            redisTemplate.setValueSerializer(new
Jackson2JsonRedisSerializer(Object.class));
            ValueOperations<String,Object> vo = redisTemplate.opsForValue();
            vo.set(key, value,time,t);
    }
```

接下来，在 UserController 内新增两个方法：一个是插入数据库数据的方法；另一个执行先查询 Redis，再查询数据库的逻辑。新增代码片段如代码清单 5-14 所示。

代码清单 5-14　Spring Boot 使用 Redis 项目 UserController 类新增方法代码

```
@GetMapping("/saveUser2")
    public User saveUser2(Long id, String userName, String userPassword){
        User user = new User(id,userName, userPassword);
        userRepository.save(user);
        return user;
    }

    @GetMapping(value = "getUser")
    public Object getUser(Long id){
        Object object =redisService.get(id.toString());
        if(object == null){
            object = (userRepository.findById(id)).get();
            if(object != null){
                redisService.set(id.toString(),object,100L,TimeUnit.SECONDS);
            }
        }
```

```
        return object;
    }
```

重启项目，先在浏览器上访问 http://localhost:8080/getUser?id=1，对数据库插入一条数据，再访问 http://localhost:8080/getUser?id=1，可以看到控制台输出如下：

```
    Hibernate: select user0_.id as id1_0_0_, user0_.user_name as user_nam2_0_0_, 
user0_.user_password as user_pas3_0_0_ from user user0_ where user0_.id=?
```

再次访问 http://localhost:8080/getUser?id=1，发现控制台没有输出，因为这次是从 Redis 内读取数据的，100 秒后，还需要从数据库查询一次。到这里，使用 Redis 缓存就完成了。还有很多可以拓展的内容，感兴趣的读者可以继续钻研。

5.3 使用 Memcached

5.3.1 Memcached 简介

Memcached 是一个自由开源的、高性能的、分布式的内存对象缓存系统。Memcached 是以 LiveJournal 旗下 Danga Interactive 公司的 Brad Fitzpatric 为首开发的一款软件。现在已成为 Mixi、Hatena、Facebook、Vox、LiveJournal 等众多服务中提高 Web 应用扩展性的重要因素。Memcached 是一种基于内存的 key-value 存储，用来存储小块的任意数据（字符串、对象）。这些数据可以是数据库调用、API 调用或者页面渲染的结果。Memcached 简洁而强大，它的简洁设计便于快速开发，降低了开发难度，解决了大数据量缓存的很多问题。它的 API 兼容大部分流行的开发语言。本质上，Memcached 是一个简洁的 key-value 存储系统。一般的使用目的是通过缓存数据库查询结果，减少数据库访问次数，以提高动态 Web 应用的速度和可扩展性。

5.3.2 配置 Memcached 依赖

新建项目，在项目中加入 Memcached 依赖，pom 文件依赖代码如代码清单 5-15 所示。

代码清单 5-15　Spring Boot 项目使用 Memcached 项目 Memcached 依赖代码

```
<dependency>
    <groupId>net.spy</groupId>
    <artifactId>spymemcached</artifactId>
    <version>2.12.2</version>
</dependency>
```

然后在配置文件配置 Memcached 信息，主要包含 Memcached 服务器地址和端口，如代码清单 5-16 所示。

代码清单 5-16　Spring Boot 项目使用 Memcached 项目配置文件代码

```
memcache.ip=localhost
memcache.port=11211
```

使用 Memcached 还需要与 Memcached 数据库建立连接来对数据库进行操作。接下来我们配置一个 MemcachedConfig，并且在配置类中封装一些 Memcached 常用的方法，如代码清单 5-17 所示。

代码清单 5-17　Spring Boot 项目使用 Memcached 项目 MemcachedConfig 类代码

```
@Component
public class MemcachedConfig implements CommandLineRunner {
    private Logger logger =  LoggerFactory.getLogger(this.getClass());

    @Value("${memcache.ip}")
    private String memcacheIp;

    @Value("${memcache.port}")
    private Integer memcachePort;

    private MemcachedClient client = null;

    @Override
    public void run(String... args) throws Exception {
        try {
            client = new MemcachedClient(new InetSocketAddress(memcacheIp,
memcachePort));
        } catch (IOException e) {
            logger.error("Connection to server failed",e);
        }
        logger.info("Connection to server success");
    }

    public MemcachedClient getClient() {
        return client;
    }


    public Boolean set(String key,int time,String value) {
        Boolean b = false;
        try{
            b=(this.getClient().set(key, time, value)).get();
        }catch (Exception e){
            logger.error(e.getMessage());
        }
        return b;
    }
```

```
    public Boolean add(String key,int time,String value){
        Boolean b = false;
        try{
            b=(this.getClient().add(key, time, value)).get();
        }catch (Exception e){
            logger.error(e.getMessage());
        }
        return b;
    }

    public Object replace(String key,int time,String value){
        Boolean b = false;
        try{
            b=(this.getClient().replace(key, time, value)).get();
        }catch (Exception e){
            logger.error(e.getMessage());
        }
        return b;
    }

    public Object append(String key,int time,String value){
        Boolean b = false;
        try{
            b=(this.getClient().append(key, value)).get();
        }catch (Exception e){
            logger.error(e.getMessage());
        }
        return b;
    }

    public Object prepend(String key,int time,String value){
        Boolean b = false;
        try{
            b=(this.getClient().prepend(key, value)).get();
        }catch (Exception e){
            logger.error(e.getMessage());
        }
        return b;
    }

    public Object cas(String key,int time,String value){
        return this.getClient().cas(key, time, value);
    }
```

```
    public Object get(String key){
        return this.getClient().get(key);
    }

    public Boolean delete(String key){
        Boolean b = false;
        try{
            b=(this.getClient().delete(key)).get();
        }catch (Exception e){
            logger.error(e.getMessage());
        }
        return b;
    }

    public long incr(String key,Integer value){
        return this.getClient().incr(key,value);
    }

    public long decr(String key,Integer value){
        return this.getClient().decr(key,value);
    }

}
```

基于对 Memcached 的常用操作大致分为如下几种。

- set 方法：当前方法用于将 value（数据值）存储在指定的 key（键）中，并可以设置对应的过期时间。如果当前 key 存在值，就更新 value；如果不存在或已经过期，就存储对应的数据值。
- add 方法：当前方法用于将 value（数据值）存储在指定的 key（键）中。如果 add 的 key 已经存在，就不会更新数据（过期的 key 会更新），之前的值将仍然保持相同，并且你将获得响应 NOT_STORED。
- replace 方法：当前方法用于替换已存在的 key（键）的 value（数据值）。如果 key 不存在，替换就会失败，并且你将获得响应 NOT_STORED。
- append 方法：当前方法用于向已存在 key（键）的 value（数据值）后面追加数据。
- prepend 方法：当前方法用于向已存在 key（键）的 value（数据值）前面追加数据。
- cas 方法：当前方法用于执行一个“检查并设置”的操作，它仅在当前客户端最后一次取值后，该 key 对应的值没有被其他客户端修改的情况下，才能够将值写入。检查是通过 cas_token 参数进行的，这个参数是 Memcached 指定给已经存在的元素的唯一的 64 位值。
- get 方法：当前方法用于获取存储在 key（键）中的 value（数据值），如果 key 不存在或者已经过期，就返回空。
- gets 方法：当前方法用于获取带有 CAS 令牌存储的 value（数据值），如果 key 不存在，就返回空。
- delete 方法：当前方法用于删除已存在的 key（键）。

- incr 方法：当前方法用于对已存在的 key（键）的数字值进行自增操作。如果 key 不存在，就返回 NOT_FOUND; 如果键的值不为数字，就返回 CLIENT_ERROR; 其他错误返回 ERROR。
- decr 方法：当前方法用于对已存在的 key（键）的数字值进行自减操作。如果 key 不存在，就返回 NOT_FOUND；如果键的值不为数字，就返回 CLIENT_ERROR；其他错误返回 ERROR。

方法指令很多，大多数都很好理解，都是一些对数据库的常用操作。这里以新增用户、查询用户、删除用户的流程进行测试，实体类还是之前的 User 实体类，实体类记得要重写 toString()方法，接下来会用到。新建一个 UserController 类，在其中加入新增查询和删除的方法，如代码清单 5-18 所示。

代码清单 5-18　Spring Boot 项目使用 Memcached 项目 UserController 类代码

```
@RestController
public class UserController {

    @Resource
    private MemcachedConfig memcachedConfig;

    @GetMapping(value = "saveUser")
    public Boolean saveUser(Long id, String userName, String userPassword){
        User user = new User(id, userName, userPassword);
        return memcachedConfig.set(id.toString(), 1000,user.toString());
    }

    @GetMapping(value = "getUserById")
    public Object getUserById(Long id) {
        return memcachedConfig.get(id.toString());
    }

    @GetMapping(value = "deleteUserCacheById")
    public Boolean deleteUserCacheById(Long id) {
        return memcachedConfig.delete(id.toString());
    }
}
```

启动项目，在浏览器上访问 http://localhost:8080/saveUser?id=1&userName=dalaoyang&userPassword=123&time=10，可以看到显示如下：

```
true
```

接下来，我们在浏览器上访问 localhost:8080/getUserById?id=1，这个方法的显示结果就是 User 实体类的 toString()方法的结果，显示如下：

```
User{id=1, userName='dalaoyang', userPassword='123'}
```

最后，我们在浏览器上访问 http://localhost:8080/deleteCacheById?id=1，删除数据，然后调用查询方法，发现返回结果为空。3 个方法到这里都测试完了，感兴趣的读者可以自己将所有方法都测试一遍，以便于具体使用。

5.3.3　使用 Memcached 缓存

使用 Memcached 作为数据库缓存的流程其实和使用 Redis 缓存一致，也是首先查询 Memcached 是否含有数据，如果数据不存在或已经过期，就先从数据库查询，再插入 Memcached 数据以提供下次使用。

配置文件与依赖文件这里就不再展示了。添加一个查询方法，以供使用 Memcached 缓存，如代码清单 5-19 所示。

代码清单 5-19　Spring Boot 项目使用 Memcached 项目 UserController 类方法代码

```
@Autowired
private UserRepository userRepository;

@GetMapping("/saveUser2")
public User saveUser2(Long id, String userName, String userPassword) {
    User user = new User(id, userName, userPassword);
    userRepository.save(user);
    return user;
}

@GetMapping(value = "getUserById2")
public Object getUserById2(Long id) {
    Object object = memcachedConfig.get(id.toString());
    if (object == null) {
        object = (userRepository.findById(id)).get();
        if (object != null) {
            memcachedConfig.set(id.toString(), 1000, object.toString());
        }
    }
    return object;
}
```

重启项目，先访问 http://localhost:8080/saveUser2?id=1&userName=dalaoyang&userPassword=123&time=10，向数据库中插入一条数据，再访问 http://localhost:8080/getUserById2?id=1，第一次控制台打印了 SQL，因为缓存没有数据，第二次控制台没有打印 SQL，因为这次查询的是 Memcached。

5.3.4 Redis 与 Memcached 的区别

对于缓存层，使用 Memcached 和 Redis 都可以完美解决，这里大致列出笔者总结的二者的区别。

1. 数据类型支持不同

Memcached 只支持简单的 key-value 存储，而 Redis 除了支持 key-value 外，还支持 list、set、hash、zset 结构。

2. 数据一致性

Memcached 内部提供了 cas 命令，可以保证在高并发下访问数据的一致性问题，而 Redis 没有提供类似 cas 的命令，但是 Redis 提供了事务的功能，可以用其保证事务的原子性。

3. value 值大小

Redis 的 value 值最大可达 1GB，而 Memcached 只有 1MB。

4. 存储方式

Memcached 将数据全部存储在内存中，当发生断电或者内存分配不足时会造成数据丢失。Redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载使用。

5. 网络 IO 模型

Redis 使用单线程的 IO 复用模型，Memcached 是多线程，非阻塞 IO 复用的网络模型。

6. 持久化支持

Redis 提供了 RDB 和 AOF 的持久化支持，Memcached 不支持持久化。

7. 应用场景

Memcached 多用于缓存数据集、临时数据、Session 等，而 Redis 除了可以用作缓存数据库外，还可以用作消息队列、数据堆栈等。

以上是笔者总结的几点区别，至于具体场景应该使用哪种，还需要具体分析。

5.4 小　结

本章实质上是对第 4 章 Spring Boot 使用数据库的扩展，利用缓存来减轻数据库的压力，进一步对 Spring Boot 的使用进行学习。

第 6 章

Spring Boot 的日志之旅

日志是追溯系统使用、问题跟踪的依据，是一个系统不可缺少的重要组成部分。在开发阶段，我们可能比较容易对系统进行监控，但是对于分布式系统，日志收集对于开发者维护系统、Bug 定位起着至关重要的作用。本章将对 Spring Boot 的一些常用日志进行分析和学习。

Spring Boot 使用 Commons Logging 进行所有内部日志记录，但保留底层日志实现。为 Java Util Logging、Log4j2 和 Logback 提供了默认配置。每种情况下，记录器都预先配置为使用控制台输出，并且提供可选的文件输出。

默认情况下，如果使用 Starters，就使用 Logback 进行日志记录。还包括适当的 Logback 路由，以确保使用 Java Util Logging、Commons Logging、Log4j 或 SLF4J 的依赖库都能正常工作。

6.1　Logback 日志

在 Spring Boot 框架中，默认使用的是 Logback 日志。接下来我们看一下 Spring Boot 是如何使用日志的。

6.1.1　Logback 简介

Logback 日志框架（官网地址：https://logback.qos.ch/）是由 Log4j 创始人开发的另一套开源日志组件。Logback 的体系非常强大，提供了 3 个模块供开发者使用。

- logback-core：属于 Logback 的基础模块，是其他两个模块的基础。
- logback-classic：可以看作 Log4j 的改进版本，同时 logback-classic 自身实现了 SLF4J API，使开发者可以在 Logback 框架与其他日志框架（如 Log4j 或 java.util.logging）之间自由切换。

- logback-access：与 Servlet 容器（如 Tomcat 和 Jetty）集成，以提供 HTTP 访问日志功能。

当然，Logback 支持开发者基于 logback-core 模块构建自定义模块。

6.1.2 日志格式

Spring Boot 项目启动后，默认可以看到如图 6-1 所示的界面。

图 6-1　Logback 项目启动日志

从图 6-1 中可以看到，Spring 的 Logo 部分是 Spring Boot 框架自带的，我们只观察日志部分。日志大致分为如下格式。

- 时间日期：显示日志打印时间，精确到毫秒。
- 日志级别：日志级别分为 FATAL、ERROR、WARN、INFO、DEBUG、TRACE。
- 进程 ID：进程 ID 指的是当前应用对应的 PID。
- 分隔符：分隔符用于区分实际日志消息的开始。
- 线程名称：括在方括号中（可能会截断控制台输出）。
- 记录器名称：一般使用类名。
- 日志内容：日志输出的具体内容。

6.1.3 控制台输出

在 Spring Boot 默认应用日志配置中，会将日志默认输出到控制台中。在默认情况下，只会记录 ERROR-level、WARN-level 和 INFO-level 级别的日志消息。当然，也可以指定日志级别进行日志输出，如果指定了日志级别，那么只会对应输出高于指定级别的日志信息。当然，Spring Boot 默认为我们提供了调试模式（建议在开发过程中开启），启动调试模式有如下两种方式。

- 启动 JAR 模式：在启动 JAR 的时候通过使用--debug 标志启动应用程序调试模式，如代码清单 6-1 所示。

代码清单 6-1　Logback 项目 jar 方式启动应用

```
java -jar myapp.jar --debug
```

- 在配置文件中的配置：在 application.properties 或者 application.yml 中配置属性 debug=true。

同时，默认日志提供彩色日志输出，如果终端支持 ANSI，那么在默认设置下，TRACE、DEBUG 和 INFO 级别为绿色，WARN 级别为黄色，ERROR 和 FATAL 级别为红色。

6.1.4　日志文件输出

默认情况下，Spring Boot 只会将日志消息打印到控制台，并不会将日志写入日志文件。但是在实际项目中，一定会需要日志文件来分析程序。其实在 Spring Boot 工程中，想要输出控制台之外的日志文件很简单，只需要在 application.properties 文件或 application.yml 文件内设置 logging.file 或 logging.path 属性即可。

- logging.file：设置日志文件，这里可以设置文件的绝对路径，也可以设置文件的相对路径，具体可以根据情况设置，如 logging.file=test.log。
- logging.path：设置日志目录，在设置好目录后，会在设置目录文件夹下创建一个 spring.log，如设置 logging.path=/Users/dalaoyang。

上述两个属性中，如果只设置一个，那么 Spring Boot 应用会默认读取该配置；如果同时设置，那么只有 logging.file 会生效。

Spring Boot 应用日志文件输出与控制台输入内容一致，在日志文件达到 10MB 的时候会自动分隔日志文件，默认情况下会记录 ERROR-level、WARN-level 和 INFO-level 消息。当然，日志文件可以通过设置 logging.file.max-size 属性更改大小限制，并非无法更改。

6.1.5　日志级别

所有受支持的日志记录系统都可以通过使用 TRACE、DEBUG、INFO、WARN、ERROR、FATAL 或 OFF 之一来在 Spring 中设置记录器级别，如下面几种格式。

- logging.level.root = WARN：root 日志以 WARN 级别输出消息。
- logging.level.com.dalaoyang = DEBUG：com.dalaoyang 包下的类以 DEBUG 级别输出。

另外，也可以设置日志组来批量设置日志级别，比如设定 com.dalaoyang.controller 和 com.dalaoyang.service 为同一组（包与包之间用英文格式的逗号分隔），如代码清单 6-2 所示。

代码清单 6-2　Logback 项目配置日志组

```
logging.group.dalaoyang =com.dalaoyang.controller,com.dalaoyang.service
```

然后，设置 dalaoyang 组日志级别为 TRACE，如代码清单 6-3 所示。

代码清单 6-3　Logback 项目配置日志组日志级别

```
logging.level.dalaoyang = TRACE
```

Spring Boot 默认提供两个日志组，如代码清单 6-4 所示。

代码清单 6-4　Logback 项目配置日志组日志级别

```
logging.group.web =org.springframework.core.codec,
org.springframework.http, org.springframework.web
   logging.group.sql =org.springframework.jdbc.core, org.hibernate.SQL
```

6.1.6　日志配置

除了上面介绍的配置属性外，其实还有很多属性供我们使用，例如：

- logging.exception-conversion-word：记录异常时使用的转换字。
- logging.file：设置日志文件。
- logging.file.max-size：最大日志文件大小。
- logging.config：日志配置。
- logging.file.max-history：最大归档文件数量。
- logging.path：日志文件目录。
- logging.pattern.console：在控制台输出的日志模式。
- logging.pattern.dateformat：日志格式内的日期格式。
- logging.pattern.file：默认使用日志模式。
- logging.pattern.level：日志级别。
- PID：当前进程 ID。

6.1.7　基于 XML 配置日志

Spring Boot 默认支持通过 XML 配置自定义日志格式及输出，并且在 ApplicationContext 创建前就已经进行了初始化。在 Spring Boot 默认使用的 Logback 中，可以通过在 src/mian/resources 文件夹下定义 logback.xml 或 logback-spring.xml 作为日志配置。

不过 Spring Boot 官方推荐优先使用带有-spring 的文件名作为你的日志配置（如使用 logback-spring.xml，而不是 logback.xml），因为如果命名为 logback-spring.xml 日志配置，就可以在日志输出的时候引入一些 Spring Boot 特有的配置项。当然，也支持自定义日志配置，比如在 application.properties 或 application.yml 中配置 logging.config=classpath:logback- config.xml，就会读取 logback-config.xml 配置对日志进行输出。

1. 控制台输出日志

接下来，我们改造一下日志文件格式。首先在 src/mian/resources 目录下创建一个 logback-spring.xml，这里以输出到控制台为例，可以在配置文件中设置如下内容，如代码清单 6-5 所示。

代码清单 6-5 Logback 项目 xml-输出到控制台

```
<configuration>
 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
   <encoder>
    <pattern>%date %-5level [%thread]  %logger{80} - %msg%n</pattern>
    <charset>UTF-8</charset>
   </encoder>
 </appender>

 <root level="debug">
    <appender-ref ref="STDOUT" />
 </root>
</configuration>
```

在上述 pattern 标签中的内容都对应日志相关的信息，分别如下。

- %date：日志输出时间，也可以使用%d 来表示，同时可以用{yyyy-MM-dd HH:mm:ss.SSS}的形式对日志的输出时间进行格式化。
- %thread：输出日志的进程名字。
- %-5level：日志级别，并且使用 5 个字符靠左对齐，也可以使用%p 输出日志级别。
- %logger{80}：日志输出者的名字。
- %msg：日志消息。
- %n：平台的换行符。
- %c：用来在日志上输出类的全名。

这里需要注意，将编码格式设置为 UTF-8，避免中文乱码。

在 root 标签内设置日志级别，效果等同于在配置文件中设置 logging.pattern.level。

2. 彩色日志输出

启动项目后，可以看到日志有对应输出，但是日志并没有颜色。接下来我们修改一下配置文件，如代码清单 6-6 所示。

代码清单 6-6 Logback 项目 xml-彩色日志配置

```
<configuration>
    <conversionRule conversionWord="clr" converterClass=
"org.springframework.boot.logging.logback.ColorConverter" />
    <conversionRule conversionWord="wex" converterClass="org.
springframework.boot.logging.logback.WhitespaceThrowableProxyConverter" />
    <conversionRule conversionWord="wEx" converterClass="org.
springframework.boot.logging.logback.
ExtendedWhitespaceThrowableProxyConverter" />
```

```
    <property name="CONSOLE_LOG_PATTERN" value="${CONSOLE_LOG_PATTERN:
-%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p})
%clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint}
%clr(%-40.40logger{39}){cyan} %clr(:){faint}
%m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}}"/>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>UTF-8</charset>
        </encoder>
    </appender>

    <root level="debug">
        <appender-ref ref="STDOUT" />
    </root>

</configuration>
```

在这里需要配置几个 Logback 提供的彩色日志类，并使用这些对日志进行修饰。这次配置有一个不同点是，Logback 配置文件可以使用 property 标签自定义属性，然后在下面使用${property 属性 name 值}。在完成上述配置后，再次启动日志就可以看到彩色日志了。

3. 日志文件输出

控制台输出日志的形式一般只有开发环境这样使用，一般来说，生产环境需要将日志输出到日志文件进行日志分析，并且会将日志根据级别输出到不同日志文件中。同时，如果日志文件太大，就可以设置日志文件根据大小分隔，如代码清单 6-7 所示。

代码清单 6-7 Logback 项目 xml-输出到日志文件

```
<configuration>
    <property name="log.path" value="/Users/dalaoyang/logs/testLog" />

    <appender name="DEBUG_FILE" class=
"ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${log.path}/log_debug.log</file>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level
%logger{50} - %msg%n</pattern>
            <charset>UTF-8</charset>
        </encoder>
        <rollingPolicy class="ch.qos.logback.core.rolling.
TimeBasedRollingPolicy">
            <fileNamePattern>${log.path}/debug/log-debug-%d{yyyy-MM-dd}.
%i.log</fileNamePattern>
```

```
            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
               <maxFileSize>50MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
            <maxHistory>30</maxHistory>
         </rollingPolicy>
         <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>debug</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
         </filter>
      </appender>

      <root level="debug">
         <appender-ref ref="DEBUG_FILE" />
      </root>

   </configuration>
```

在上述配置中包含很多新标签，分别说明如下。

- file：日志文件位置。
- maxFileSize：设置最大日志文件大小。
- maxHistory：只保留最近 30 天的日志，防止日志过多占用磁盘。
- fileNamePattern：指定精确到分的日志切分方式。
- filter：标签中的 level 设置日志级别。

4. 输出指定包文件日志

Logback 可以指定输出某个包下的类的日志，这种方式比较简单，只需要指定包路径及日志级别即可，如代码清单 6-8 所示。

代码清单 6-8　Logback 项目 xml-输出到日志文件

```
<logger name="com.dalaoyang" level="DEBUG">
   <appender-ref ref="DEBUG_FILE" />
</logger>
```

Spring Boot 使用 Logback 日志大致就这几种形式。当然，可以根据具体项目更加细化地配置日志文件，这里不再更多地描述了，毕竟实际业务场景不同，配置的方法也不同。

6.2　Log4j 日志

Log4j 是笔者第一个接触的日志框架，截至目前还有很多工程使用这个日志框架。接下来我们学习一下 Spring Boot 如何使用 Log4j 日志。

6.2.1 Log4j 简介

Log4j 日志（官网地址：http://logging.apache.org/log4j/1.2/）是一个使用 Java 编写的日志框架，由 Apache Software Foundation 的一个专门的 Committers 团队开发。

Log4j 虽然是一个使用 Java 开发的框架，但是它同样支持很多编程语言使用，如 C、C++、.Net、PL/SQL。

6.2.2 Spring Boot 使用 Log4j

虽然现在已经不推荐使用 Log4j 了，但是这里还是简单介绍一下 Spring Boot 如何使用 Log4j 框架。在 6.1 节我们了解到 Spring Boot 默认引用 Logback 日志，这里需要在 pom 文件中移除 spring-boot-starter-logging 依赖，并加入 Log4j 依赖，如代码清单 6-9 所示。

代码清单 6-9　Log4j 项目-pom 文件配置

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j</artifactId>
    <version>1.3.2.RELEASE</version>
    <type>pom</type>
</dependency>
```

6.2.3 控制台输出

在使用 Log4j 的时候，默认会读取 src/main/resources 下的 log4j.properties 文件。其实道理和配置 Logback 类似，先进行输出到控制台的配置，如代码清单 6-10 所示。

代码清单 6-10　Log4j 项目-pom 文件配置

```
log4j.rootLogger=debug,CONSOLE
```

```
##输出到控制台
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=DEBUG
log4j.appender.CONSOLE.layout.ConversionPattern=%d{yyyy-MM-dd HH\:mm\:ss}
-%-4r [%t] %-5p  %x - %m%n
log4j.appender.CONSOLE.Target=System.out
log4j.appender.CONSOLE.Encoding=UTF-8
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
```

上述配置大致包含如下内容。

- log4j.appender.CONSOLE：控制台日志输出类。
- log4j.appender.CONSOLE.Threshold：日志级别。
- log4j.appender.CONSOLE.layout.ConversionPattern：日志输出信息格式。
- log4j.appender.CONSOLE.Target：使用 System.error 作为输出。
- log4j.appender.CONSOLE.Encoding：日志编码格式。
- log4j.appender.CONSOLE.layout：设置输出样式。

在日志输出格式中可以使用如下自定义样式进行配置。

- %c：输出所属的类目，通常就是所在类的全名。
- %C：输出 Logger 所在类的名称，通常就是所在类的全名。
- %d：输出日志时间点的日期或时间，默认格式为 ISO8601，与 Logback 一致，可以格式化日期格式，比如%d{yyy MMM dd HH:mm:ss}。
- %F：输出所在类的类名称，只有类名。
- %l：输出语句所在的行数，包括类名、方法名、文件名、行数。
- %L：输出语句所在的行数。
- %m：输出代码中指定的信息，如 log(message)中的 message。
- %M：输出方法名。
- %p：输出日志级别，即 DEBUG、INFO、WARN、ERROR、FATAL。
- %r：输出自应用启动到输出该 Log 信息耗费的毫秒数。
- %t：输出产生该日志事件的线程名。
- %n：输出一个回车换行符，Windows 平台为“/r/n”，UNIX 平台为“/n”。
- %%：用来输出百分号“%”。

6.2.4　日志文件输出

如果需要使用日志输出，就需要对配置文件新增如下配置，如代码清单 6-11 所示。

代码清单 6-11　Log4j 项目-Log4j 配置文件

```
log4j.rootLogger=debug,FILE
```

```
log4j.appender.FILE=org.apache.log4j.DailyRollingFileAppender
log4j.appender.FILE.File=/Users/dalaoyang/Downloads/log
log4j.appender.FILE.DatePattern = '_'yyyy-MM-dd-HH-mm'.log'
log4j.appender.FILE.MaxFileSize=10MB
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=%d%n%m%n
```

上述配置大致包含如下内容：

- log4j.appender.FILE：日志文件输出类。
- log4j.appender.FILE.File：日志文件输出路径。
- log4j.appender.FILE.DatePattern：日志文件后缀格式。
- log4j.appender.FILE.MaxFileSize：日志文件输出大小。
- log4j.appender.FILE.layout：设置输出样式。
- log4j.appender.FILE.layout.ConversionPattern：日志输出信息格式。

到这里，已经配置完成了。感兴趣的读者可以自己配置看看。Log4j 官网上已经不推荐使用 Log4j 框架，而推荐使用拥有更好性能的 Log4j 2 框架，所以这里稍作了解即可。

6.3 Log4j 2 日志

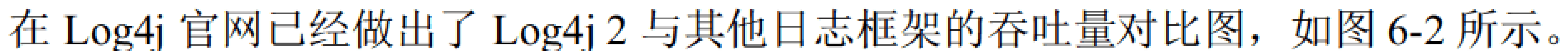
Apache Log4j 2（官网地址：https://logging.apache.org/log4j/2.x/index.html）是对 Log4j 的升级，它对前身 Log4j 1.x 进行了重大改进。6.2 节我们对 Log4j 框架有了一定的了解，接下来学习 Spring Boot 对 Log4j 2 框架的使用。

6.3.1 Log4j 2 简介

Log4j 1.x 版本虽然已经广泛使用于很多应用程序中，然而，经过多年的发展，它的发展速度已经变缓。由于 Log4j 1.x 需要符合 Java 的旧版本并且在 2015 年 8 月已经寿终正寝，因此维护变得更加困难。Log4j 1.x 的替代方案 SLF4J/Logback 对框架进行了许多必要的改进。为什么要使用 Log4j 2 呢？下面是 Log4j 官网给出的解释。

- Log4j 2 被设计为可以作为审计框架使用。Log4j 1.x 和 Logback 都会在重新配置的时候失去事件，而 Log4j 2 不会。在 Logback 中，Appender 中的异常对应用从来都是不可见的，但 Log4j 2 的 Appender 可以设置为允许将异常渗透给应用程序。
- Log4j 2 包含基于 LMAX Disruptor 库的下一代异步日志器。在多线程的情况下，异步日志器具有比 Log4j 1.x 和 Logback 高出 10 倍的吞吐性能以及更低的延迟。
- Log4j 2 在稳定记录状态下，对单机应用是无垃圾的，对 Web 应用是低垃圾的。这不仅降低了垃圾回收器的压力，还可以提供更好的响应性能。

- Log4j 2 使用插件系统，使得它非常容易通过新的 Appender、Filter、Layout、Lookup 和 Pattern Converter 来扩展框架，且不需要对 Log4j 做任何修改。
- 由于插件系统的配置更简单了，因此配置项不需要声明类名称。
- 支持自定义日志级别。自定义日志级别可以在代码或配置中定义。
- 支持 Lambda 表达式。运行在 Java 8 上的客户端代码可以使用 Lambda 表达式实现仅在对应的日志级别启用时延迟构造日志消息。由于不需要明确地层层把关，因此带来了更简洁的代码。
- 支持 Message 对象。Message 支持感兴趣或复杂的结构体在日志系统中传输，且可以被高效地操作。用户可以自由地创建他们自己的 Message 类型，并编写自定义的 Layout、Filter 和 Lookup 来操作它们。
- Log4j 1.x 支持 Appender 上的 Filter。Logback 引入了 TurboFilter 在事件被 Logger 处理之前对它们进行过滤。Log4j 2 支持的 Filter 可以设置为在被 Logger 接管之前就处理事件，如同它在 Logger 或 Appender 中被处理。
- 很多 Logback 的 Appender 不接受一个 Layout，且只能发送固定格式的数据。而大多数 Log4j 2 的 Appender 接受 Layout，允许数据以任意所需的格式传输。
- Log4j 1.x 和 Logback 中的 Layout 返回一个 String，这可能导致一些编码问题。Log4j 2 使用更简单的方法，Layout 总是返回一个字节数组。优点是这意味着它们可以用于任何 Appender，而不仅仅是写入 OutputStream 中的那些。
- Syslog Appender 既支持 TCP 又支持 UDP，同样支持 BSD 系统日志以及 RFC 5424 格式。
- Log4j 2 利用了 Java 5 的并发优势，并且尽可能最低程度上进行锁定。Log4j 1.x 中已知存在死锁问题。其中很多已经在 Logback 中修复，但很多 Logback 的 class 文件仍然需要在更高的编译级别中同步。


在 Log4j 官网已经做出了 Log4j 2 与其他日志框架的吞吐量对比图，如图 6-2 所示。

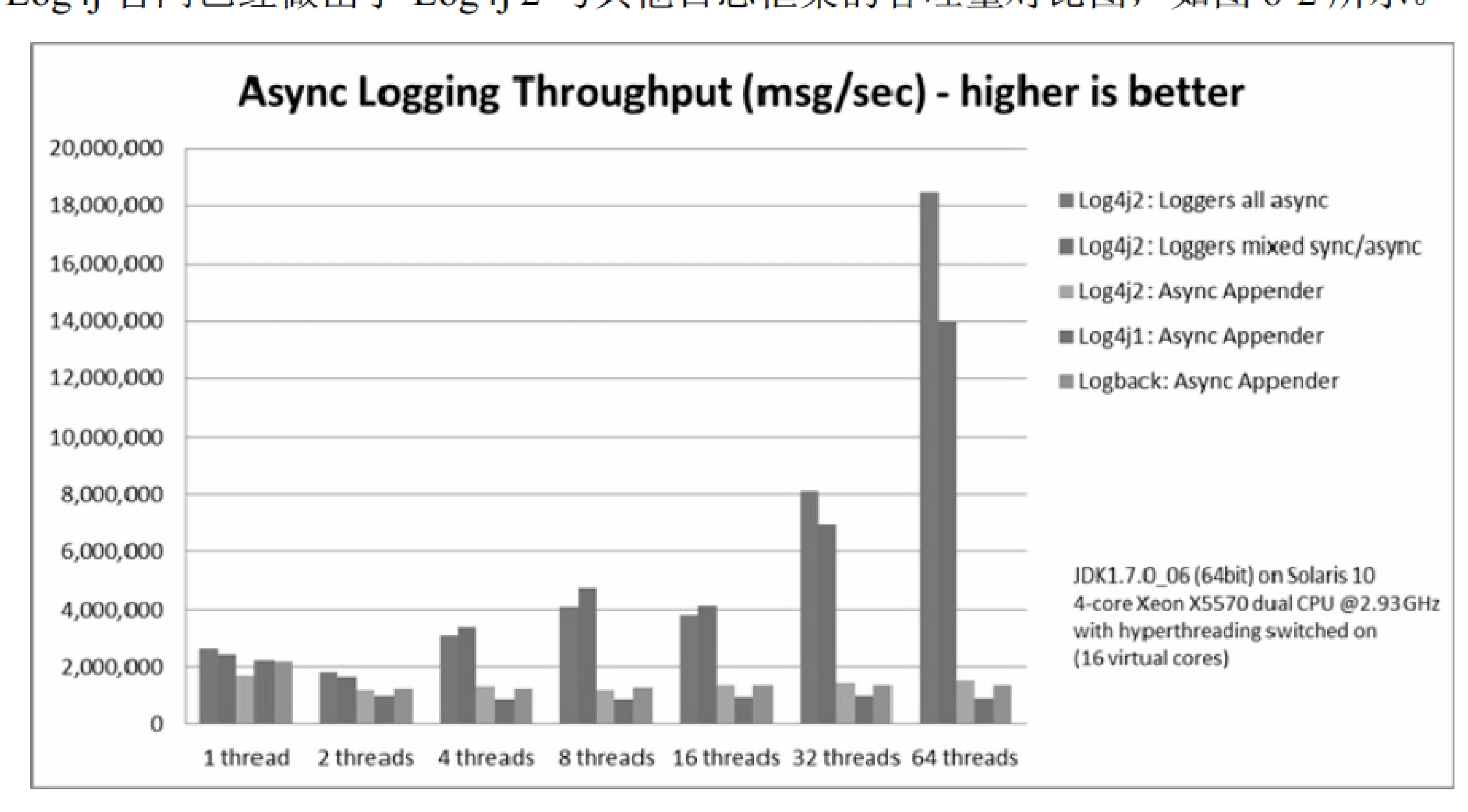


图 6-2　Log4j 2 与其他日志框架的异步吞吐量对比图

从图 6-2 中可以看到，Log4j 2 框架与其他框架相比，线程数越多，吞吐量越大，性能优势十分明显。

接下来，我们再看一下官网上 Log4j 2 与其他日志框架的响应时间对比图，如图 6-3 所示。

图 6-3　Log4j 2 与其他日志框架的响应时间对比图

图 6-3 比较了 Logback 1.1.7、Log4j 1.2.17 基于 ArrayBlockingQueue 的异步 Appender 与 Log4j 2.6 提供的异步日志记录的各种选项的响应时间延迟。在每秒 128 000 个消息的工作负载下，使用 16 个线程（以每秒 8 000 个消息的速率进行记录），我们看到 Logback 1.1.7、Log4j 1.2.17 遇到的延迟峰值比 Log4j 2 大几个数量级。

6.3.2　Spring Boot 使用 Log4j 2

在 Spring Boot 中使用 Log4j 2 与使用 Log4j 的过程类似。首先，需要在 pom 文件中排除默认日志框架并引入 Log4j 2 依赖，这里额外加入了 Disruptor 依赖，用于解决 Log4j2 日志版本较低报错的问题，如代码清单 6-12 所示。

代码清单 6-12　Log4j 2 项目-Log4j 2 依赖文件

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

```
<dependency>
        <groupId>com.lmax</groupId>
        <artifactId>disruptor</artifactId>
        <version>3.4.2</version>
    </dependency>
```

6.3.3　控制台输出

我们先来看一下如何使用 Log4j 在控制台输出日志，首先在 src/main/resources 文件夹下创建 log4j2-spring.xml 文件，如代码清单 6-13 所示。

代码清单 6-13　Log4j 2 项目-日志文件输出到控制台

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Properties>
        <Property name="LOG_PATTERN">%d{yyyy-MM-dd HH:mm:ss:SSS} - %-5level
- %pid - %t - %c{1.}:%L - %m%n</Property>
    </Properties>
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT" follow="true">
            <ThresholdFilter level="trace" onMatch="ACCEPT" onMismatch=
"DENY" />
            <PatternLayout pattern="${LOG_PATTERN}"/>
        </Console>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

内容很好理解，其中：

- Property：用于配置自定义属性，name 是属性名称。
- Appenders：用于定义输出日志类型。
- Console ：用于配置输出到控制台的配置。
- ThresholdFilter：定义输出的日志级别。
- PatternLayout：输出日志的格式。
- Loggers：在这里引入 Appenders 才能使对应 Appenders 生效。
- AppenderRef：定义生效的 Appenders。

6.3.4 日志文件输出

接下来我们学习如何将日志输出到日志文件，如代码清单 6-14 所示。

代码清单 6-14 Log4j 2 项目-输出日志到日志文件

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Properties>
        <Property name="LOG_PATTERN">%d{yyyy-MM-dd HH:mm:ss:SSS} - %-5level
- %pid - %t - %c{1.}:%L - %m%n</Property>
        <Property name="FILE_PATH">/Users/dalaoyang/logs/</Property>
    </Properties>
    <Appenders>
        <File name="File" fileName="${FILE_PATH}sys.log">
            <PatternLayout>
                <pattern>${LOG_PATTERN}</pattern>
            </PatternLayout>
        </File>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="File" />
        </Root>
    </Loggers>
</Configuration>
```

这里的配置与输出到控制台大致相同，只不过将 Appenders 标签内的 Console 标签替换为 File 标签。

6.3.5 异步日志

在 Log4j 官网建议开发者查看日志的方式改为异步方式，这样的好处在于可以使用单独的线程来打印日志，提高日志效率，避免由于打印日志而影响业务功能。

1. AsyncAppender 方式

官网中是这样介绍 AsyncAppender 的：AsyncAppender 是通过引用别的 Appender 来实现的，当有日志事件到达时，会开启另一个线程来处理它们。需要注意的是，如果在 Appender 的时候出现异常，对应用来说是无法感知的。AsyncAppender 应该在它引用的 Appender 之后配置，默认使用 java.util.concurrent.ArrayBlockingQueue 实现，不需要其他外部的类库。当使用此 Appender 的时候，在多线程的环境下需要注意，阻塞队列容易受到锁争用的影响，这可能会对性能产生影响。这时，我们应该考虑使用无锁的异步记录器（AsyncLogger）。

接下来先使用 AsyncAppender 的方式使用异步日志，如代码清单 6-15 所示。

代码清单 6-15　Log4j2 项目-AsyncAppender 输出日志

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Properties>
        <Property name="LOG_PATTERN">%d{yyyy-MM-dd HH:mm:ss:SSS} - %-5level
- %pid - %t - %c{1.}:%L - %m%n</Property>
        <Property name="FILE_PATH">/Users/dalaoyang/logs/</Property>
    </Properties>
    <Appenders>
        <File name="File" fileName="${FILE_PATH}sys.log">
            <PatternLayout>
                <pattern>${LOG_PATTERN}</pattern>
            </PatternLayout>
        </File>
        <Async name="ASYNC">
            <AppenderRef ref="File"/>
        </Async>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="ASYNC"/>
        </Root>
    </Loggers>
</Configuration>
```

2. AsyncLogger 方式

AsyncLogger 是官方推荐的异步方式，它提供了两种方式使用异步日志，即全局异步和混合异步。全局异步是指所有的日志都进行异步的日志记录，而混合异步是指可以同时使用同步日志和异步日志。

下面是全局日志的方式，配置文件如代码清单 6-16 所示。

代码清单 6-16　Log4j 2 项目-AsyncLogger 输出日志

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Properties>
        <Property name="LOG_PATTERN">%d{yyyy-MM-dd HH:mm:ss:SSS} - %-5level
- %pid - %t - %c{1.}:%L - %m%n</Property>
        <Property name="FILE_PATH">/Users/dalaoyang/logs/</Property>
    </Properties>
    <Appenders>
```

```
        <RandomAccessFile name="RandomAccessFile" fileName=
"${FILE_PATH}async.log" immediateFlush="false" append="false">
            <PatternLayout>
                <Pattern>%d %p %c{1.} [%t] %m %ex%n</Pattern>
            </PatternLayout>
        </RandomAccessFile>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="RandomAccessFile"/>
        </Root>
    </Loggers>
</Configuration>
```

在系统初始化的时候需要增加全局配置，如代码清单 6-17 所示。

代码清单 6-17　Log4j 2 项目-全局异步日志参数配置

```
System.setProperty("log4j2.contextSelector,
"org.apache.logging.log4j.core.async.AsyncLoggerContextSelector");
```

你可以在第一次获取 Logger 之前设置，也可以选择在加载 JVM 启动参数里设置，如代码清单 6-18 所示。

代码清单 6-18　Log4j 2 项目-全局异步日志 JVM 启动参数配置

```
java
-Dog4j2.contextSelector=org.apache.logging.log4j.core.async.AsyncLoggerConte
xtSelector
```

接下来看一下混合异步的模式，如代码清单 6-19 所示。

代码清单 6-19　Log4j 2 项目-混合异步日志参数配置

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Properties>
        <Property name="LOG_PATTERN">%d{yyyy-MM-dd HH:mm:ss:SSS} - %-5level
- %pid - %t - %c{1.}:%L - %m%n</Property>
        <Property name="FILE_PATH">/Users/dalaoyang/logs/</Property>
    </Properties>
    <Appenders>
        <RandomAccessFile name="RandomAccessFile" fileName=
"${FILE_PATH}async.log" immediateFlush="false" append="false">
            <PatternLayout>
                <Pattern>%d %p %c{1.} [%t] %m %ex%n</Pattern>
```

```
            </PatternLayout>
        </RandomAccessFile>
    </Appenders>
    <Loggers>
        <AsyncLogger name="com.springboot" level="info" includeLocation="true">
            <AppenderRef ref="RandomAccessFile"/>
        </AsyncLogger>
        <Root level="info">
            <AppenderRef ref="RandomAccessFile"/>
        </Root>
    </Loggers>
</Configuration>
```

在上述配置中，Root 的 Logger 是同步日志，而 com.springboot 的 Logger 是异步日志。

Log4j 2 提供了很优秀的异步日志供我们使用，尽管这样，我们也没有必要盲目追求，毕竟稳定的才是最好的。

6.4 ELK 日志收集

在 6.1~6.3 小节中介绍了关于 Spring Boot 的几种流行的日志框架，但在微服务的场景下，每个服务都部署在不同的服务器中，如果每次排查问题都需要挨个服务器查看日志，就太麻烦了，所以我们需要对日志进行集中收集，然后统一查看所有日志。

当前比较流行的日志收集有很多，本节以 ELK 为例（Elasticsearch+Logstash+Kibana）介绍如何进行日志收集。

6.4.1 ELK 日志收集流程介绍

简单的 ELK 日志收集流程如下：

（1）在微服务服务器上部署 Logstash，对日志文件进行数据采集，将采集到的数据输出到 Elasticsearch 集群中。

（2）Kibana 读取 Elasticsearch 数据，提供 Web 展示页面。

6.4.2 ELK 安装

接下来介绍 ELK 的安装步骤。

1. Java

Elasticsearch、Logstash 和 Kibana 都需要在 Java 环境下运行，所以需要提前在服务器或者物理机安装 Java，如果需要，可以查看本书安装 Java 的过程，这里不再赘述。

2. Elasticsearch

登录 Elasticsearch 官网，下载 Elasticsearch 压缩包（下载地址：https://www.elastic.co/cn/downloads/elasticsearch）。这里以 Linux 安装为例，首先下载压缩包，这里下载 6.5.4 版本，如代码清单 6-20 所示。

代码清单 6-20　ELK 项目-下载 Elasticsearch 代码

```
    wget https://artifacts.elastic.co/downloads/ elasticsearch/
elasticsearch-6.5.4.tar.gz
```

下载完成后解压文件，如代码清单 6-21 所示。

代码清单 6-21　ELK 项目-解压 Elasticsearch 代码

```
    tar -zxvf elasticsearch-6.5.4.tar.gz
```

3. Logstash

无论是什么系统，首先在 Logstash 官网下载 Logstash 压缩包（下载地址：https://www.elastic.co/cn/downloads/logstash），如代码清单 6-22 所示。

代码清单 6-22　ELK 项目-下载 Logstash 代码

```
    wget https://artifacts.elastic.co/downloads/logstash/logstash-6.5.4.tar.gz
```

下载完成后解压文件，如代码清单 6-23 所示。

代码清单 6-23　ELK 项目-解压 Logstash 代码

```
    tar -zxvf logstash-6.5.4.tar.gz
```

4. Kibana

到 Kibana 官网下载 Kibana 压缩包（下载地址：https://www.elastic.co/cn/ downloads/kibana），还是以 Linux 安装为例，过程与 Logstash 类似，如代码清单 6-24 所示。

代码清单 6-24　ELK 项目-下载 Kibana 代码

```
    wget https://artifacts.elastic.co/downloads/kibana/
kibana-6.5.4-linux-x86_64.tar.gz
```

解压 Kibana 压缩包，如代码清单 6-25 所示。

代码清单 6-25　ELK 项目-解压 Kibana 代码

```
    tar -zxvf kibana-6.5.4-linux-x86_64.tar.gz
```

6.4.3　ELK 配置

首先启动 Elasticsearch，进入 Elasticsearch 安装目录的 bin 目录下，执行如下命令启动 Elasticsearch，如代码清单 6-26 所示。

代码清单 6-26　ELK 项目-启动 Elasticsearch

```
./elasticsearch
```

这里以收集本机/Users/dalaoyang/logs/sys.log 日志文件为例，首先查看日志文件内容，如图 6-4 所示。

```
sys.log
2018-12-01 21:27:33:326 - INFO   - 3099 - restartedMain - o.s.w.s.h.SimpleUrlHandlerMapping:373 - Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-12-01 21:27:33:643 - INFO   - 3099 - restartedMain - o.s.w.s.m.m.a.RequestMappingHandlerAdapter:574 - Looking for @ControllerAdvice: org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@16c237: startup date [Sat Dec 01 21:27:29 CST 2018]; root of context hierarchy
2018-12-01 21:27:33:770 - INFO   - 3099 - restartedMain - o.s.w.s.m.m.a.RequestMappingHandlerMapping:547 - Mapped "{[/],methods=[GET]}" onto public java.lang.String com.dalaoyang.Chapter63Application.index()
2018-12-01 21:27:33:781 - INFO   - 3099 - restartedMain - o.s.w.s.m.m.a.RequestMappingHandlerMapping:547 - Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController.error(javax.servlet.http.HttpServletRequest)
2018-12-01 21:27:33:783 - INFO   - 3099 - restartedMain - o.s.w.s.m.m.a.RequestMappingHandlerMapping:547 - Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
2018-12-01 21:27:33:823 - INFO   - 3099 - restartedMain - o.s.w.s.h.SimpleUrlHandlerMapping:373 - Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-12-01 21:27:33:825 - INFO   - 3099 - restartedMain - o.s.w.s.h.SimpleUrlHandlerMapping:373 - Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-12-01 21:27:34:151 - INFO   - 3099 - restartedMain - o.s.b.d.a.OptionalLiveReloadServer:57 - LiveReload server is running on port 35729
2018-12-01 21:27:34:194 - INFO   - 3099 - restartedMain - o.s.j.e.a.AnnotationMBeanExporter:433 - Registering beans for JMX exposure on startup
2018-12-01 21:27:34:216 - INFO   - 3099 - restartedMain - o.a.c.h.Http11NioProtocol:180 - Starting ProtocolHandler ["http-nio-8080"]
2018-12-01 21:27:34:249 - INFO   - 3099 - restartedMain - o.a.t.u.n.NioSelectorPool:180 - Using a shared selector for servlet write/read
2018-12-01 21:27:34:271 - INFO   - 3099 - restartedMain - o.s.b.w.e.t.TomcatWebServer:206 - Tomcat started on port(s): 8080 (http) with context path ''
2018-12-01 21:27:34:390 - INFO   - 3099 - restartedMain - c.d.Chapter63Application:59 - Started Chapter63Application in 5.605 seconds (JVM running for 7.896)
2018-12-01 23:11:52:732 - INFO   - 3099 - Thread-22 - o.s.b.w.s.c.AnnotationConfigServletWebServerApplicationContext:993 - Closing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@16c237: startup date [Sat Dec 01 21:27:29 CST 2018]; root of context hierarchy
2018-12-01 23:11:52:765 - INFO   - 3099 - Thread-22 - o.s.j.e.a.AnnotationMBeanExporter:451 - Unregistering JMX-exposed beans on shutdown
```

图 6-4　ELK 项目测试日志文件截图

1. 配置 Kibana

进入 Kibana 安装目录下的 config 目录，打开 kibana.yml，添加 Elasticsearch 配置，如代码清单 6-27 所示。

代码清单 6-27　ELK 项目-Kibana 配置信息

```
#Elasticsearch 主机地址
elasticsearch.url: "http://localhost:9200"
#允许远程访问
server.port: "0.0.0.0"
```

然后进入 Kibana 安装目录下的 bin 目录，输出如下命令启动 Kibana，如代码清单 6-28 所示。

代码清单 6-28　ELK 项目-启动 Kibana

```
./kibana
```

启动后控制台如图 6-5 所示。

```
[ x ▶ /usr/local/Cellar/kibana/6.5.3 ▶ bin/kibana
  log   [10:40:19.354] [info][status][plugin:kibana@6.5.3] Status changed from uninitialized to green - Ready
  log   [10:40:19.392] [info][status][plugin:elasticsearch@6.5.3] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [10:40:19.728] [info][status][plugin:timelion@6.5.3] Status changed from uninitialized to green - Ready
  log   [10:40:19.738] [info][status][plugin:console@6.5.3] Status changed from uninitialized to green - Ready
  log   [10:40:19.745] [info][status][plugin:metrics@6.5.3] Status changed from uninitialized to green - Ready
  log   [10:40:19.955] [info][status][plugin:elasticsearch@6.5.3] Status changed from yellow to green - Ready
  log   [10:40:19.981] [info][listening] Server running at http://localhost:5601
```

图 6-5　ELK 项目测试日志文件截图

从提示中可以看到，访问 http://localhost:5601 可以查看 Kibana 的 Web 页面，如图 6-6 所示。



图 6-6　ELK Kibana 首页

2. 配置 Logstash

进入 Logstash 目录，Logstash 一般读取 conf 结尾的配置文件，所以创建一个 log2es.conf，如代码清单 6-29 所示。

代码清单 6-29　ELK 项目-Logstash 配置信息

```
#从日志文件读取数据
#file{}
#type 日志类型
#path 日志位置
#           可以直接读取文件（a.log）
#           可以读取所有后缀为 log 的日志（*.log）
#           读取文件夹下所有文件（路径）
#start_position 文件读取开始位置 （beginning）
#sincedb_path 从什么位置读取（设置为/dev/null 自动从开始位置读取）
input {
    file {
      type => "sys-log"
```

```
        path => ["/Users/dalaoyang/logs/sys.log"]
        start_position => "beginning"
        sincedb_path => "/dev/null"
      }
  }

  #数据的输出指向了 es 集群
  #hosts Elasticsearch 主机地址
  #index Elasticsearch 索引名称
  output {
    elasticsearch {
      hosts => "localhost:9200"
      index => "test-log-%{+YYYY.MM.dd}"
    }
  }
```

配置文件内容解释可以查看代码清单中的注释，这里就不再介绍了。接下来启动 Logstash，进入 Logstash 安装目录下的 bin 目录，输入启动命令，如代码清单 6-30 所示。

代码清单 6-30　ELK 项目-启动 Logstash 命令

```
./logstash -f ../log2es.conf
```

6.4.4　使用 Kibana 查看日志

在 ELK 三者都启动后日志已经在收集了，打开 Kibana 系统管理，可以看到刚刚创建的索引 test-log-2019.01.19，如图 6-7 所示。

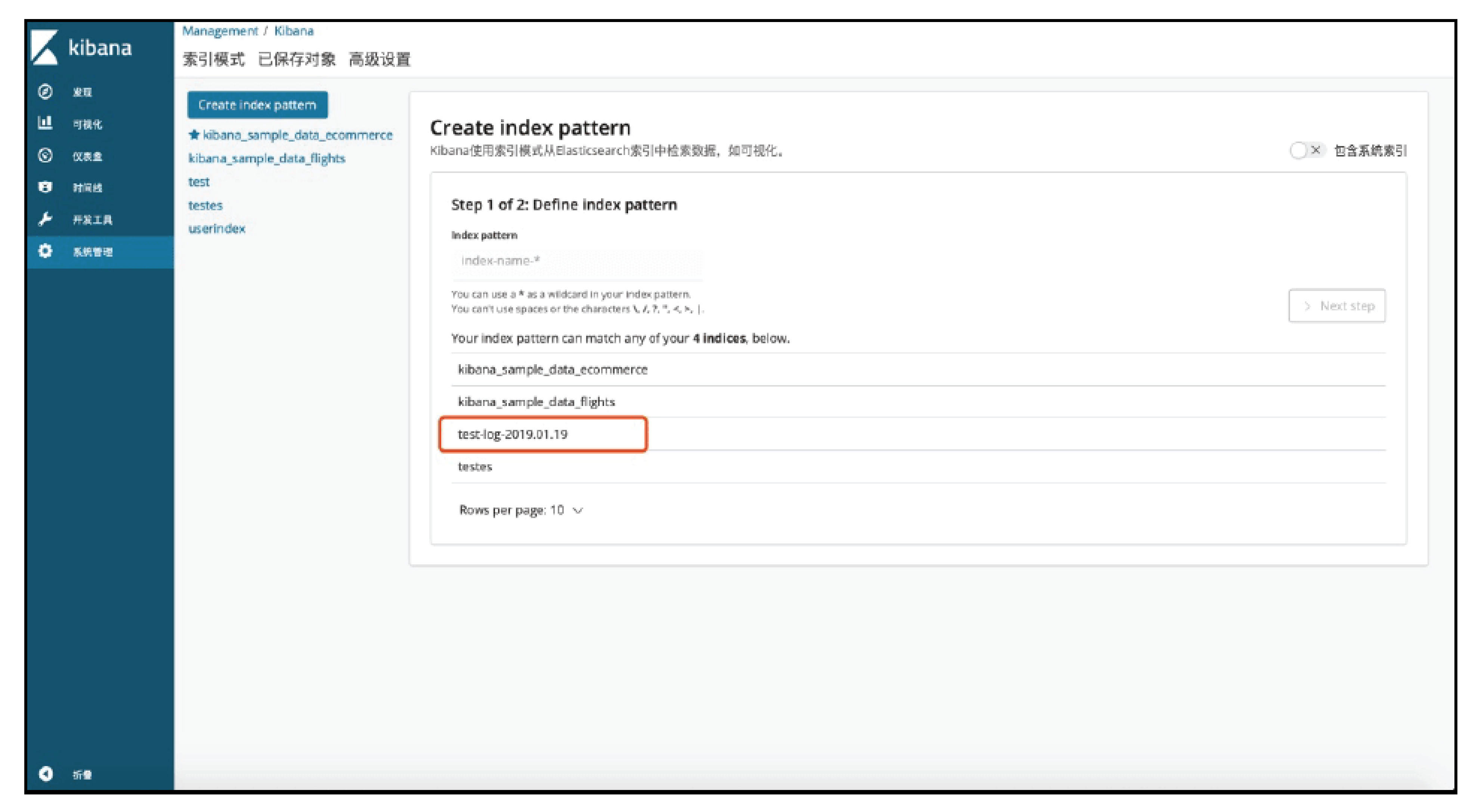


图 6-7　ELK Kibana 关联索引页 1

接下来在 Index pattern 输入框中输入索引名称 test-log-2019.01.19（这里可以使用通配符*等），然后单击 Next step 按钮，显示如图 6-8 所示的页面。

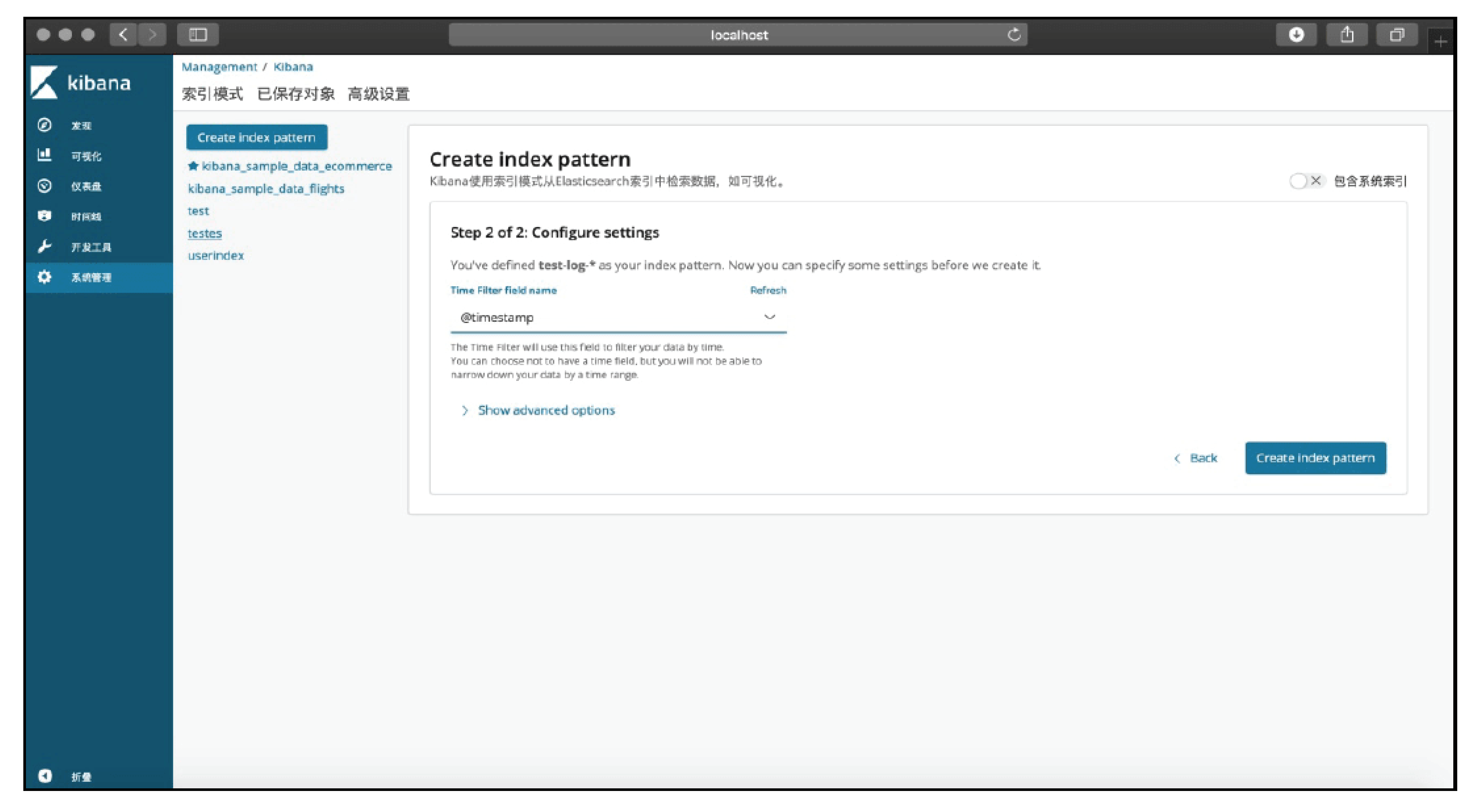


图 6-8　ELK Kibana 关联索引页 2

单击 Create index pattern 按钮创建索引，显示如图 6-9 所示的页面。

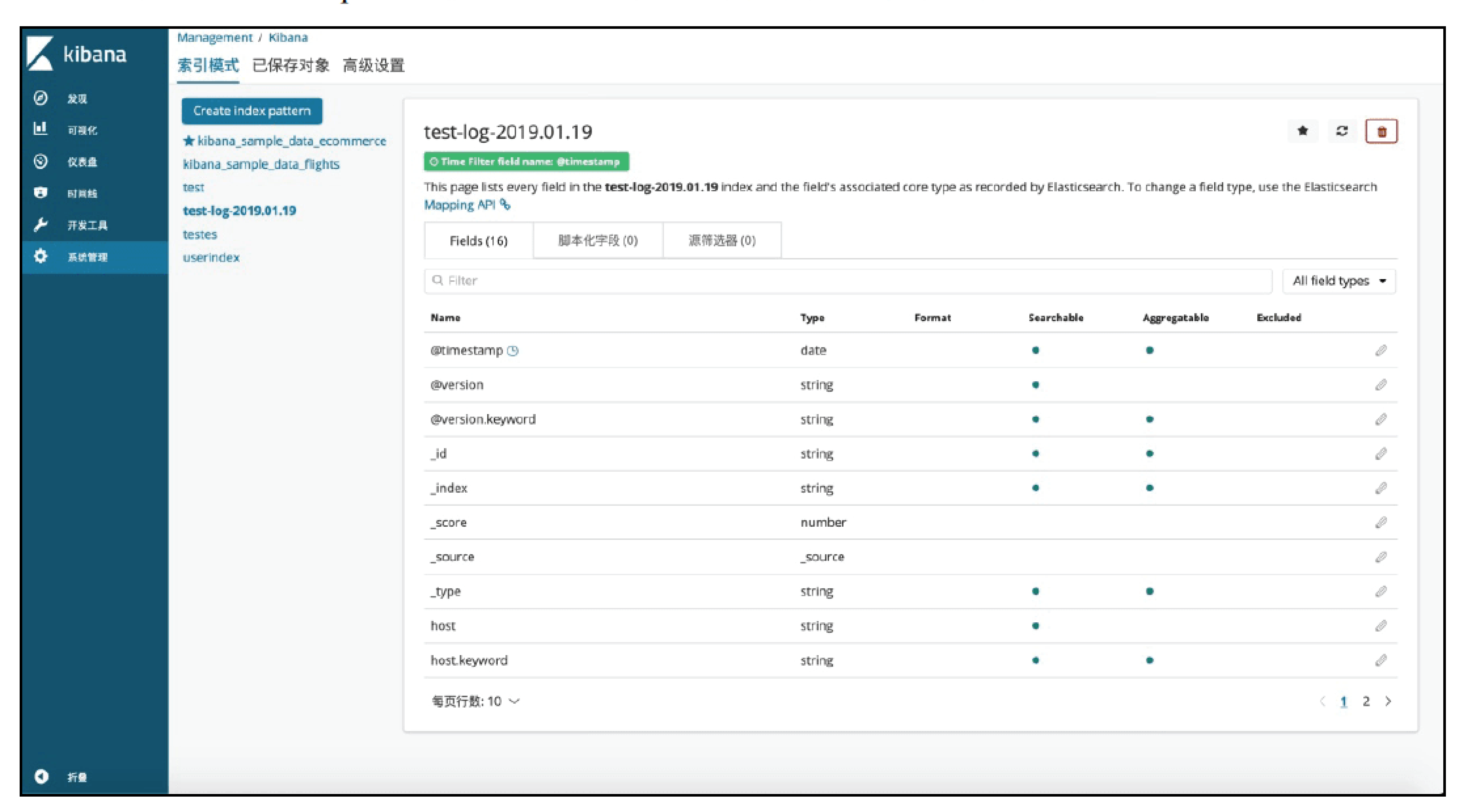


图 6-9　ELK Kibana 关联索引页　3

在如图 6-9 所示的页面中可以查看当前索引中的字段，因为这里只是简单使用，所以没有太多字段，如果需要，可以使用 Logstash 自定义字段。接下来，单击菜单栏中的“发现”按钮，然后选择索引 test-log-2019.01.19，可以看到我们要收集的日志内容已经收集进来了，如图 6-10 所示。

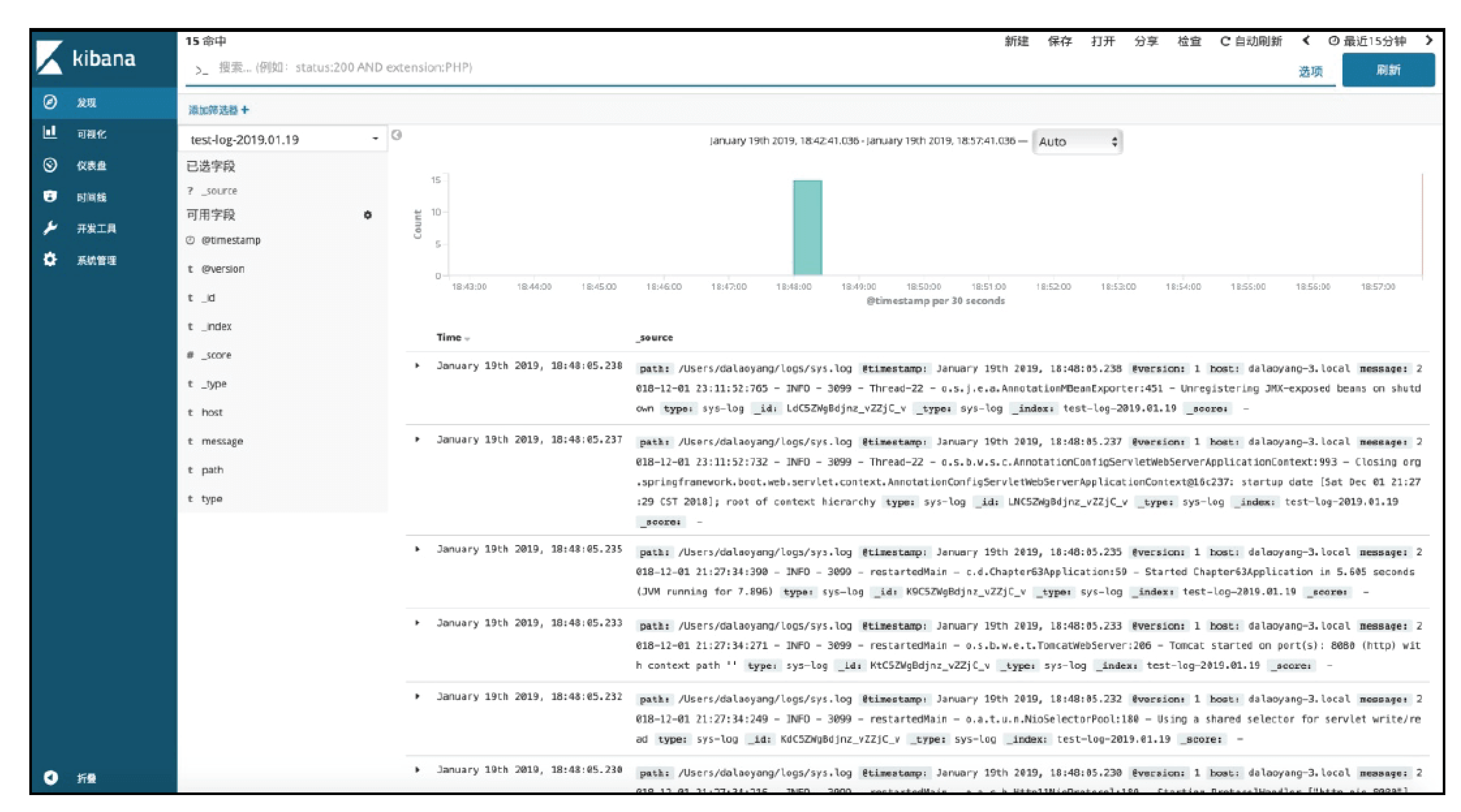


图 6-10　ELK Kibana 索引查看页 1

接下来测试日志收集，在日志文件中最下面一行追加如下内容：

```
测试日志收集。
```

再来查看 Kibana，页面如图 6-11 所示。



图 6-11　ELK Kibana 索引查看页 2

从图 6-11 中可以看到，倒数第二条就是刚刚在日志文件中输出的内容，由于笔者一不小心多按了一次回车键，将回车键显示的空白数据也收集到了。

6.4.5 Spring Boot 直接输出到 Logstash

前面介绍了使用 Logstash 对日志文件进行收集，其实也可以将 Spring Boot 应用程序直接远程输出到 Logstash。

这里以 Logback 日志为例，新建项目，在项目中加入 Logstash 依赖，如代码清单 6-31 所示。

代码清单 6-31　ELK 项目- Logstash 依赖

```
<dependency>
        <groupId>net.logstash.logback</groupId>
        <artifactId>logstash-logback-encoder</artifactId>
        <version>5.3</version>
</dependency>
```

接下来，在 src/resources 目录下创建 logback-spring.xml 配置文件，在配置文件中将对日志进行格式化，并且输出到控制台和 Logstash。需要注意的是，在 destination 属性中配置的地址和端口要与 Logstash 输入源的地址和端口一致，比如这里使用的是 127.0.0.1:4560，则在 Logstash 输入源中要与这个配置一致。其中 logback-spring.xml 内容如代码清单 6-32 所示。

代码清单 6-32　ELK 项目-logback-spring.xml 文件内容

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>

    <appender name="LOGSTASH" class="net.logstash.logback.appender.
LogstashTcpSocketAppender">
        <destination>127.0.0.1:4560</destination>
        <!-- 日志输出编码 -->
        <encoder charset="UTF-8"
                 class="net.logstash.logback.encoder.
LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp>
                    <timeZone>UTC</timeZone>
                </timestamp>
                <pattern>
                    <pattern>
                        {
                        "logLevel": "%level",
                        "serviceName": "${springAppName:-}",
                        "pid": "${PID:-}",
                        "thread": "%thread",
                        "class": "%logger{40}",
```

```
                    "rest": "%message"
                    }
                </pattern>
            </pattern>
        </providers>
    </encoder>
</appender>

<root level="INFO">
    <appender-ref ref="LOGSTASH" />
    <appender-ref ref="CONSOLE" />
</root>


</configuration>
```

为了方便起见，这里的 Logstash 只是将日志输出到控制台，配置文件内容如代码清单 6-33 所示。

代码清单 6-33　ELK 项目-Logstash 配置文件内容

```
input {
  tcp {
    mode => "server"
    host => "127.0.0.1"
    port => 4560
    codec => json_lines
  }
}
output {
   stdout {codec => rubydebug}
}
```

启动 Logstash，然后启动应用程序，查看 Logstash 控制台即可看到日志输出，这里就不进行演示了。

6.4.6　ELK 日志收集优化方案及建议

ELK 的简单使用到这里就结束了。如果数据特别多，上述方案就会为 Elasticsearch 带来很大的压力。为了缓解 Elasticsearch 的压力，可以将 Logstash 收集的内容不直接输出到 Elasticsearch 中，而是输出到缓冲层，比如 Redis 或者 Kafka，然后使用一个 Logstash 从缓冲层输出到 Elasticsearch。当然，还有很多种方案进行日志收集，比如使用 Filebeat 替换 Logstash 等。笔者在生产环节搭建过 ELK 配置，这里提几点建议：

（1）根据日志量判断 Elasticsearch 的集群选择，不要盲目追求高可用，实际应用需要根据实际场景的预算等因素使用。

（2）缓冲层选择，一般来说选择 Kafka 和 Redis。虽然 Kafka 作为日志消息很适合，具备高吞吐量等，但是如果需求不是很大，并且环境中不存在 Kafka，就没有必要使用 Kafka 作为消息缓存层，使用现有的 Redis 也未尝不可。

（3）内存分配，ELK 三者部署都是占有一定内存的，并且官网建议的配置都很大。建议结合场景来修改配置，毕竟预算是很重要的一环。

6.5 小　　结

本章对 Spring Boot 的几种日志框架进行了学习，并且了解了常用的日志收集方法，相信经过这一章的学习会让读者对日志的了解有一定的提高，从而对系统维护等有更好的手段。

第 7 章

Spring Boot 的安全之旅

安全是每一个应用都必须面对的问题，一个应用如果没有设置好安全框架，那么很容易被别人利用，进而做一些非法的事情。我们可以做一个这样的假设，比如应用中没有安全框架，意味着所有人可以进行所有操作，这样对于一些后台系统来说，就会造成很大程度的风险。还有很多种原因，安全框架就这样诞生。本章将介绍常用的两个 Java 安全框架：Apache Shiro 和 Spring Security，并且使用 Spring Boot 结合二者进行简单的权限控制使用（本章的案例只是将 Spring Boot 结合安全框架进行认证和授权，由于安全框架的强大及功能性的复杂，因此并没有过多地介绍，如果基础很好，或者对安全框架很了解，可以直接跳过本章）。

7.1　使用 Shiro 安全管理

Shiro 是由 Apache 开源的一款强大的安全框架，本节从了解 Shiro 框架开始，带领大家学习 Spring Boot 如何使用 Shiro 进行身份认证和权限认证。

7.1.1　什么是 Shiro

Apache Shiro（官网地址：http://shiro.apache.org/）是一个功能强大且易于使用的 Java 安全框架，可以利用它进行身份验证、授权、加密和会话管理。通过使用 Shiro 易于理解的 API 文档，可以轻松地构建任何应用程序。

如 Apache Shiro 官网所说，Apache Shiro 的首要目标是易于使用和理解。安全有时可能非常复杂，甚至是痛苦的，但并非必须如此。框架应尽可能掩盖复杂性，并提供简洁直观的 API，以简化开发人员的工作，并确保其应用程序安全地工作。

以下是 Apache Shiro 可以做的一些事情：

- 验证用户身份。
- 为用户执行访问控制，例如确定是否为用户分配了某个安全角色或确定是否允许用户执行某些操作。
- 在任何环境中使用 Session API，即使没有 Web 容器或 EJB 容器也是如此。
- 在身份验证、访问控制或会话生命周期内对事件做出反应。
- 聚合用户安全数据的一个或多个数据源，并将其全部显示为单个复合用户“视图”。
- 启用单点登录（SSO）功能。
- 无须登录即可为用户关联启用“记住我”服务。

Apache Shiro 是一个具有许多功能的综合应用程序安全框架，如图 7-1 所示。

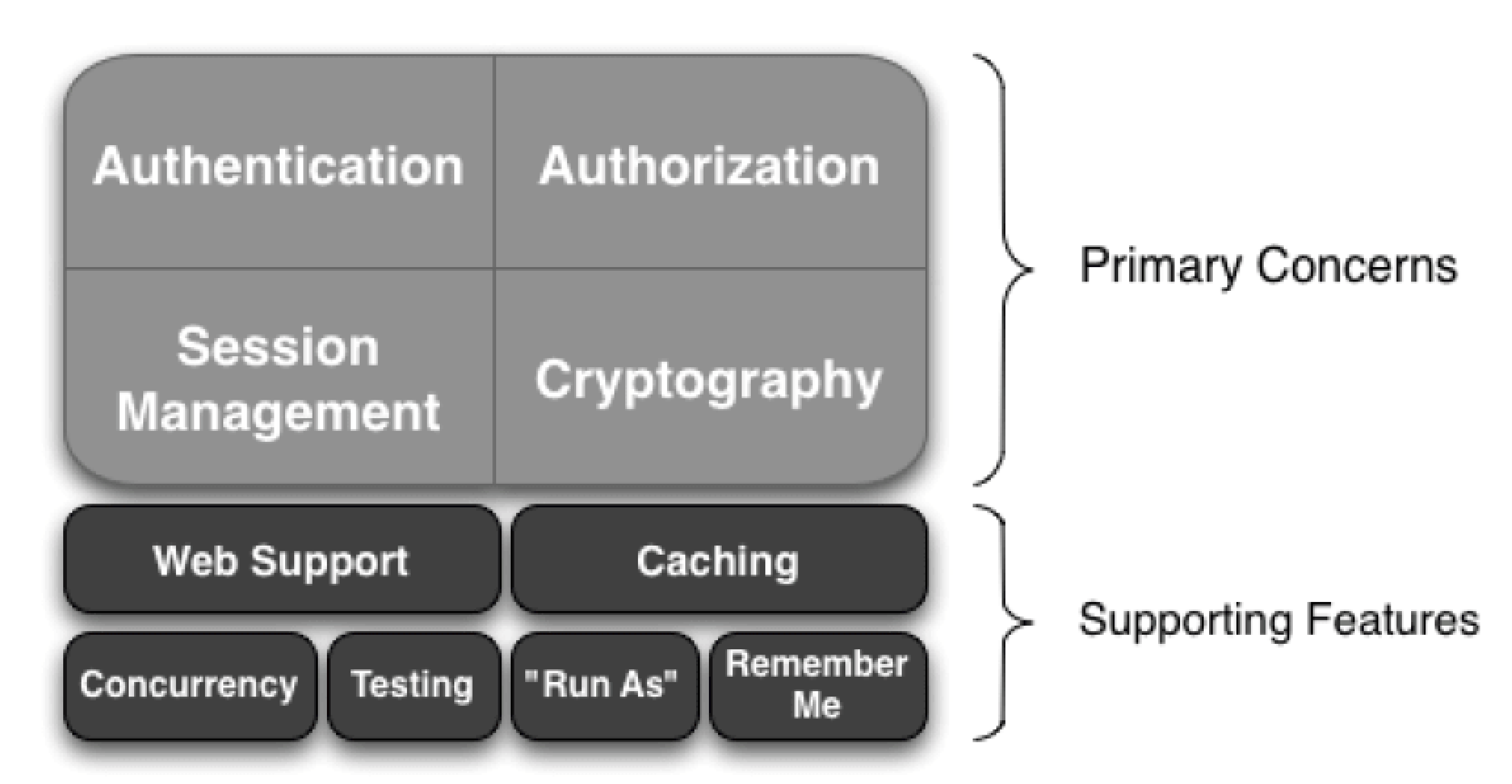


图 7-1 Apache Shiro 功能图

Shiro 提供了 Shiro 开发团队所称的“应用程序安全的 4 大基石”——身份验证、授权、会话管理和加密。

- 身份认证：其实身份认证可以理解为“登录”。
- 授权：授权是指一些权限的认证，比如管理员可以访问所有页面，但是普通用户只能访问部分页面。
- 会话管理：可以理解为 Shiro 为我们管理用户的会话（如 Session）。
- 加密：使用加密算法来保证数据的安全。

以上是 4 个主要的功能，如图 7-1 所示，还提供了其他功能，分别说明如下。

- Web 支持：Shiro 的 Web 支持 API 可帮助用户轻松保护 Web 应用程序。
- 缓存：Shiro 提供了缓存，可以确保安全操作保持快速高效。
- 并发：Apache Shiro 支持具有并发功能的多线程应用程序。
- 测试：存在测试支持以帮助用户编写单元和集成测试，并确保代码按预期受到保护。
- 运行方式：允许用户假定其他用户的身份（如果允许）的功能，有时在管理方案中很有用。
- 记住我：记住用户在会话中的身份，这样他们只需要在强制要求时登录。

7.1.2　使用 Shiro 做权限控制

刚刚介绍了 Apache Shiro 的基本功能，接下来带领大家学习 Spring Boot 如何使用 Shiro 框架进行身份认证和权限管理。

1. 场景及数据库介绍

在创建项目之前，先介绍一下需要实现的场景，数据库表设计如图 7-2 所示。

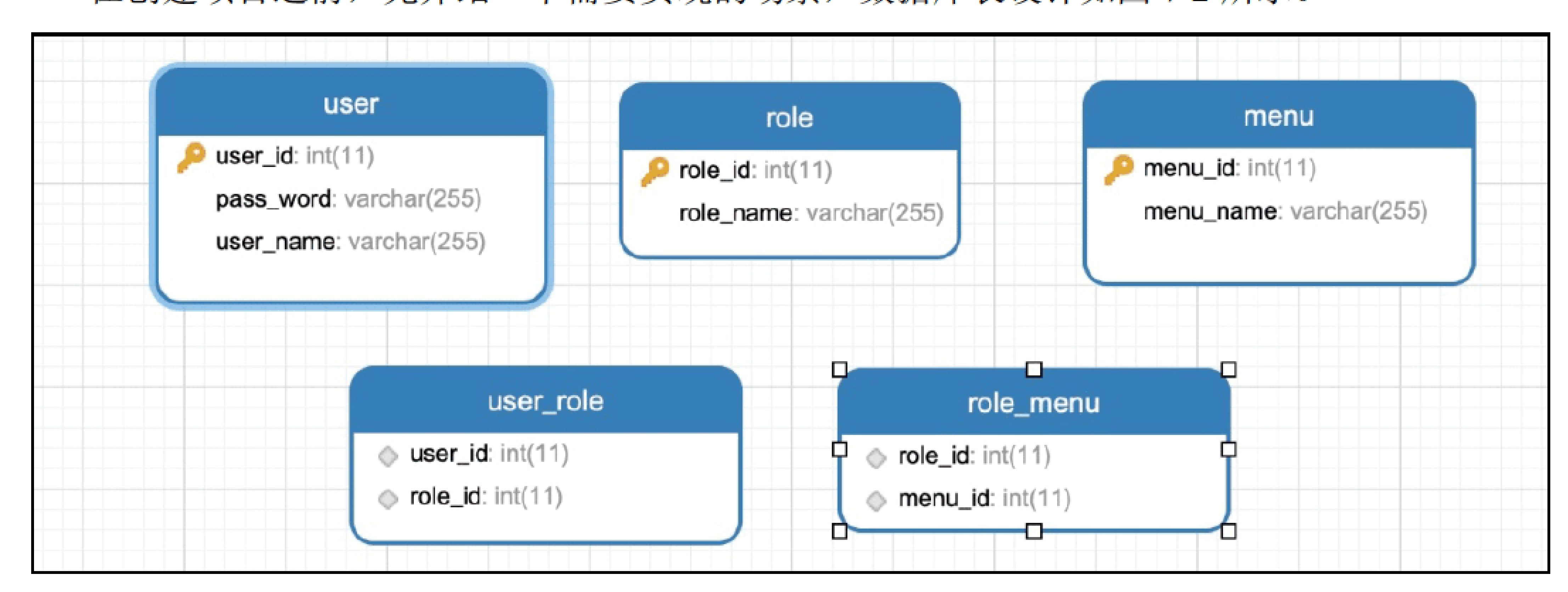


图 7-2　Shiro 项目数据库设计图

其中分为两种角色：admin 和 user，如果用户角色为 admin，则可以进行 4 个菜单的请求（add、delete、update 和 select，这里只有 select 和 delete），如果用户角色为 user，则只可以进行 select 请求。如果没有权限，就会跳转到 401 页面，index 页面可以不登录访问。为了方便，默认插入了两个用户：dalaoyang 有 admin 权限；xiaoli 有 user 权限。插入数据脚本如代码清单 7-1 所示。

代码清单 7-1　Shiro 项目数据库默认数据插入 SQL

```
    INSERT INTO `menu`(`menu_id`, `menu_name`) VALUES (1, 'add');
    INSERT INTO `menu`(`menu_id`, `menu_name`) VALUES (2, 'delete');
    INSERT INTO `menu`(`menu_id`, `menu_name`) VALUES (3, 'update');
    INSERT INTO `menu`(`menu_id`, `menu_name`) VALUES (4, 'select');
    INSERT INTO `role`(`role_id`, `role_name`) VALUES (1, 'admin');
    INSERT INTO `role`(`role_id`, `role_name`) VALUES (2, 'user');
    INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (1, 1);
    INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (1, 2);
    INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (1, 3);
    INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (1, 4);
    INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (2, 4);
    INSERT INTO `user`(`user_id`, `pass_word`, `user_name`) VALUES (1, '123',
'dalaoyang');
    INSERT INTO `user`(`user_id`, `pass_word`, `user_name`) VALUES (2, '123',
'xiaoli');
```

```
INSERT INTO `user_role`(`role_id`, `user_id`) VALUES (1, 1);
INSERT INTO `user_role`(`role_id`, `user_id`) VALUES (2, 2);
```

2. 依赖配置

接下来我们新建一个项目，由于这里需要使用数据库，因此加入了 MySQL 和 JPA 的依赖，模板框架使用的是 Thymeleaf，同时加入 Shiro 依赖，如代码清单 7-2 所示。

代码清单 7-2　Shiro 项目依赖文件代码

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring</artifactId>
    <version>1.4.0</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>net.sourceforge.nekohtml</groupId>
    <artifactId>nekohtml</artifactId>
    <version>1.9.15</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

配置文件这里不再赘述，都是关于数据库和 JPA 的配置，如需查阅，可以在本书源码中查看。

3. 实体类及数据操作层

结合上述场景，可以看出 user 表和 role 表的关系是多对多，role 表和 menu 表的关系也是多对多，理解了关系，创建实体类就比较容易了。首先创建一个 User 实体，使用@ManyToMany 表明是多对多的关系，在@JoinTable 注解中注明中间表的表名以及关联两个表的字段，如代码清单 7-3 所示。

代码清单 7-3　Shiro 项目数据库 User 实体类代码

```
@Entity
public class User implements Serializable {

    @Id
    @GeneratedValue
    private Integer userId;
    private String userName;
    private String passWord;

    @ManyToMany(fetch= FetchType.EAGER)
    @JoinTable(name = "UserRole", joinColumns = {@JoinColumn(name = "userId")},
           inverseJoinColumns ={@JoinColumn(name = "roleId") })
    private List<Role> roleList;

    ... //这里省略 set、get 方法
}
```

接下来创建 Role 实体。和 User 实体类似，分别注明与 User 和 Menu 的多对多关系，如代码清单 7-4 所示。

代码清单 7-4　Shiro 项目 Role 实体类代码

```
@Entity
public class Role implements Serializable {

    @Id
    @GeneratedValue
    private Integer roleId;
    private String roleName;

    @ManyToMany(fetch= FetchType.EAGER)
    @JoinTable(name="RoleMenu",joinColumns={@JoinColumn(name="roleId")},
inverseJoinColumns={@JoinColumn(name="menuId")})
    private List<Menu> menuList;

    @ManyToMany
    @JoinTable(name="UserRole",joinColumns={@JoinColumn(name="roleId")},
inverseJoinColumns={@JoinColumn(name="userId")})
    private List<User> userList;

    ... //这里省略 set、get 方法
}
```

最后是 Menu 实体，这里表明与 Role 的多对多关系，如代码清单 7-5 所示。

代码清单 7-5　Shiro 项目 Menu 实体类代码

```
@Entity
public class Menu  implements Serializable {

    @Id
    @GeneratedValue
    private Integer menuId;
    private String menuName;

    @ManyToMany
    @JoinTable(name="RoleMenu",joinColumns={@JoinColumn(name="menuId")},
inverseJoinColumns={@JoinColumn(name="roleId")})
    private List<Role> roleList;

    ... //这里省略 set、get 方法
}
```

创建一个 JPA 数据操作层，里面加入一个根据用户名查询用户的方法，如代码清单 7-6 所示。

代码清单 7-6　Shiro 项目 UserRepository 类代码

```
public interface UserRepository extends JpaRepository<User,Long> {
    User findByUserName(String username);
}
```

4. Shiro 配置

创建一个 ShiroConfig，然后创建一个 shiroFilter 方法。在 Shiro 使用认证和授权时，其实都是通过 ShiroFilterFactoryBean 设置一些 Shiro 的拦截器进行的，拦截器会以 LinkedHashMap 的形式存储需要拦截的资源及链接，并且会按照顺序执行，其中键为拦截的资源或链接，值为拦截的形式（比如 authc:所有 URL 都必须认证通过才可以访问，anon:所有 URL 都可以匿名访问），在拦截的过程中可以使用通配符，比如/**为拦截所有，所以一般/**放在最下面。同时，可以通过 ShiroFilterFactoryBean 设置登录链接、未授权链接、登录成功跳转页等，这里设置的 shiroFilter 方法内容如代码清单 7-7 所示。

代码清单 7-7　Shiro 项目 ShiroConfig 类 shiroFilter 方法代码

```
@Bean
public ShiroFilterFactoryBean shiroFilter(SecurityManager securityManager){
        ShiroFilterFactoryBean shiroFilterFactoryBean = new
ShiroFilterFactoryBean();
        shiroFilterFactoryBean.setSecurityManager(securityManager);
        //shiro 拦截器
        Map<String,String> filterChainDefinitionMap = new
LinkedHashMap<String,String>();
```

```
        //<!-- authc:所有 URL 都必须认证通过才可以访问,anon:所有 URL 都可以匿名访问-->
        //<!-- 过滤链定义，从上向下顺序执行，一般将/**放在最下面 -->

        // 配置不被拦截的资源及链接
        filterChainDefinitionMap.put("/static/**", "anon");
        // 退出过滤器
        filterChainDefinitionMap.put("/logout", "logout");

        //配置需要认证权限
        filterChainDefinitionMap.put("/**", "authc");
        // 默认寻找登录链接
        shiroFilterFactoryBean.setLoginUrl("/login");
        // 登录成功后要跳转的链接
        shiroFilterFactoryBean.setSuccessUrl("/index");

        //未授权的跳转链接
        shiroFilterFactoryBean.setUnauthorizedUrl("/401");
        shiroFilterFactoryBean.setFilterChainDefinitionMap
(filterChainDefinitionMap);
        return shiroFilterFactoryBean;
    }
```

同时，需要在 ShiroConfig 类中开启 shiro aop 注解支持，如果没有开启，权限验证就会失效，如代码清单 7-8 所示。

代码清单 7-8　Shiro 项目 AuthorizationAttributeSourceAdvisor 方法代码

```
    @Bean
    public AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor(SecurityManager securityManager){
        AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor = new
AuthorizationAttributeSourceAdvisor();
        authorizationAttributeSourceAdvisor.setSecurityManager
(securityManager);
        return authorizationAttributeSourceAdvisor;
    }
```

接下来创建一个方法处理一些异常信息，如代码清单 7-9 所示。

代码清单 7-9　Shiro 项目 createSimpleMappingExceptionResolver 方法代码

```
    @Bean(name="simpleMappingExceptionResolver")
    public SimpleMappingExceptionResolver
createSimpleMappingExceptionResolver() {
        SimpleMappingExceptionResolver simpleMappingExceptionResolver = new
SimpleMappingExceptionResolver();
```

```
        Properties mappings = new Properties();
        //数据库异常处理
        mappings.setProperty("DatabaseException", "databaseError");
        //未经过认证
        mappings.setProperty("UnauthorizedException","401");
        // None by default
        simpleMappingExceptionResolver.setExceptionMappings(mappings);
        // No default
        simpleMappingExceptionResolver.setDefaultErrorView("error");
        // Default is "exception"
        simpleMappingExceptionResolver.setExceptionAttribute("ex");
        return simpleMappingExceptionResolver;
    }
```

最后，我们需要在 ShiroConfig 内设置自定义身份认证的 Realm，完整 ShiroConfig 类代码可在本书源代码中查看。MyShiroRealm 类代码如代码清单 7-10 所示。

代码清单 7-10　Shiro 项目 MyShiroRealm 类代码

```
@Configuration
public class MyShiroRealm extends AuthorizingRealm {

    @Resource
    private UserRepository userRepository;

    //授权方法，主要用于获取角色的菜单权限
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principals) {
        SimpleAuthorizationInfo authorizationInfo = new
SimpleAuthorizationInfo();
        User userInfo  = (User)principals.getPrimaryPrincipal();
        for(Role role:userInfo.getRoleList()){
            authorizationInfo.addRole(role.getRoleName());
            for(Menu menu:role.getMenuList()){
                authorizationInfo.addStringPermission(menu.getMenuName());
            }
        }
        return authorizationInfo;
    }

    //认证方法，主要用于校验用户名和密码
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo
(AuthenticationToken token)
```

```
            throws AuthenticationException {
        //获得当前用户的用户名
        String username = (String)token.getPrincipal();
        //根据用户名查询用户
        User user = userRepository.findByUserName(username);
        if(user == null){
            return null;
        }
        //校验用户名、密码是否正确
        SimpleAuthenticationInfo authenticationInfo = new
SimpleAuthenticationInfo(
                user,
                user.getPassWord(),
                getName()
        );
        return authenticationInfo;
    }
}
```

其中，doGetAuthorizationInfo 方法用于授权，doGetAuthenticationInfo 方法用于验证用户信息，也就是我们常说的登录。

5. 前端页面

本案例中场景设计为 5 个页面，分别是 401 页面、delete 页面、index 页面、login 页面及 Select 页面 401 页面的代码。如代码清单 7-11 所示。

代码清单 7-11　Shiro 项目 401 页面代码

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>401</title>
</head>
<body>
401
</body>
</html>
```

delete 页面的代码如代码清单 7-12 所示。

代码清单 7-12　Shiro 项目 delete 页面代码

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
delete
</body>
</html>
```

index 页面中设置了一个注销按钮，如代码清单 7-13 所示。

代码清单 7-13　Shiro 项目 index 页面代码

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
index
<br/>
<form th:action="@{/logout}" method="post">
    <p><input type="submit" value="注销"/></p>
</form>
</body>
</html>
```

login 页面通过表单提交数据，如代码清单 7-14 所示。

代码清单 7-14　Shiro 项目 login 页面代码

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
<body>
错误信息：<h4 th:text="${msg}"></h4>
<form action="" method="post">
    <p>账号：<input type="text" name="username" value="dalaoyang"/></p>
    <p>密码：<input type="text" name="password" value="123"/></p>
    <p><input type="submit" value="登录"/></p>
</form>
```

```
</body>
</html>
```

最后是 select 页面，代码如代码清单 7-15 所示。

代码清单 7-15　Shiro 项目 select 页面代码

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
select
</body>
</html>
```

6. Controller

最后需要创建 Controller 进行页面跳转，@RequiresPermissions 注解设置 select 方法应该有 select 权限，@RequiresRoles 注解设置 delete 方法需要有 admin 的角色。其实 Shiro 提供了如下几个注解供使用。

- @RequiresAuthentication：表示当前已经通过了身份认证，即 Subject. isAuthenticated()返回 true。
- @RequiresUser：表示当前用户已经通过身份验证或者通过“记住我”登录的。
- @RequiresRoles：可以通过属性值 value 设置角色，角色可以设置一个或者多个，并且使用 logical 属性指定角色需要同时包含多个权限还是只包含一个权限。比如@RequiresRoles (value={"admin", "user"}, logical= Logical.AND)为当前需要用户同时包含 admin 和 user 权限。
- @RequiresPermissions：与上面的注解类似，判断用户是否含有菜单权限，属性值与 @RequiresRoles 一致。
- @RequiresGuest：表明当前用户没有通过身份验证或通过“记住我”登录过，也就是游客身份。

接下来，我们看一下 ShiroController 的代码，需要注意 login 方法中根据 HttpServletRequest 获取 Shiro 处理的异常信息来给出一些提示，比如用户名不存在或者密码错误。完整代码如代码清单 7-16 所示。

代码清单 7-16　Shiro 项目 ShiroController 类代码

```
@Controller
public class ShiroController {
    @GetMapping({"/","/index"})
    public String index(){
```

```
            return"index";
        }

        @GetMapping("/401")
        public String unauthorizedRole(){
            return "401";
        }

        @GetMapping("/delete")
        @RequiresRoles("admin")
        public String delete(){
            return "delete";
        }

        @GetMapping("/select")
        @RequiresPermissions("select")
        public String select(){
            return "select";
        }

        @RequestMapping("/login")
        public String login(HttpServletRequest request, Map<String, Object> map){
            // 如果登录失败，就从 HttpServletRequest 中获取 shiro 处理的异常信息，获取
shiroLoginFailure 就是 shiro 异常类的全名
            String exception = (String) request.getAttribute ("shiroLoginFailure");
            String msg = "";
            //根据异常判断错误类型
            if (exception != null) {
                if (UnknownAccountException.class.getName().equals(exception)) {
                    msg = "用户名不存在！";
                } else if (IncorrectCredentialsException.class.getName().
equals(exception)) {
                    msg = "密码错误！";
                } else {
                    msg = exception;
                }
            }
            map.put("msg", msg);
            return "/login";
        }

        @GetMapping("/logout")
        public String logout(){
```

```
            return "/login";
        }

    }
```

7. 测试

到这里，项目就已经配置完成了。启动项目，这里简单介绍一下笔者用来测试的方法。

（1）在不登录的情况下可以访问 index 和 login 页面，访问 select 页面和 delete 页面会跳转到 login 页面。

（2）使用 dalaoyang 用户登录的话，可以在登录后访问任意页面。

（3）使用 xiaoli 用户登录的话，除了访问 delete 页面会跳转到 401 页面以外，访问其他页面都会正常跳转。

通过以上测试，完全可以测试出 Shiro 框架做到了认证及授权。读者也可以使用其他方式进行测试，如果读者对 Shiro 感兴趣，可以在此基础上进行扩展，使用更多的功能。

7.2　使用 Spring Security

Spring Security（官网地址：https://spring.io/projects/spring-security）是 Spring 家族的安全框架，本节将介绍使用 Spring Boot 结合 Spring Security 进行身份认证和权限认证。

7.2.1　Spring Security 简介

Spring Security 是一个能够为基于 Spring 的企业应用系统提供声明式的安全访问控制解决方案的安全框架。它提供了一组可以在 Spring 应用上下文中配置的 Bean，充分利用了 Spring IoC（Inversion of Control，控制反转）、DI（Dependency Injection，依赖注入）和 AOP（面向切面编程）功能，为应用系统提供声明式的安全访问控制功能，减少了为企业系统安全控制编写大量重复代码的工作。

Spring Security 提供了非常多强大且常用的功能。

- 身份认证：Spring Security 提供了多粒度的身份认证，最熟悉的就是我们常用的登录功能。
- 授权：通俗地说，授权是指权限认证。当你向服务器发起一个请求时，服务器会对你的权限进行验证，如果权限不足，就可能重定向到特定的界面或返回 HTTP 响应码。
- 加密：Spring Security 提供了多种加密方式供我们使用（如 SHA、MD5 或 Bcrypt），可以根据需求选择。
- 会话管理：Spring Security 拥有特殊的会话管理机制，会对会话进行保护、定期检测会话是否超时等。

- Session 管理：如果有需要的话，那么可以配置 Spring Security 检测无效的 Session ID 提交并将用户重定向到一个指定的 URL。
- 支持 HTTP/HTTPS：支持服务器同时使用 HTTP 和 HTTPS，如果需要特殊 URL，那么只能使用 HTTPS，可以在配置中直接配置进行使用。
- 支持 Basic 和 Digest 认证：Basic（基本）验证和 Digest（摘要）验证是在 Web 应用中流行的替代身份验证机制。基本验证是指在客户端请求时，提供用户名和密码验证的一种方式，常见的如 EUREKA。摘要认证属于基本认证的升级版，将传输的密码加密，解决了 HTTP 基本认证不安全的问题。
- Remember-Me：Remember-Me 身份验证是指网站能够记住一个主体的身份之间的会话。当第一次登录时，服务器发送 cookie 给浏览器，浏览器将 cookie 保存一段时间，当再次访问时，如果在会话中发现 cookie，则进行自动登录。
- 提供 CSRF 解决方案：CSRF 是 Cross Site Request Forgery 的缩写，CSRF 是指跨站请求伪造。是一种通过伪装成受信任用户的请求来利用受信任的网站。Spring Security 提供了解决这个问题的方案。
- CORS：Spring Security 提供了对 CORS（跨域资源共享）的支持。
- 安全 HTTP 响应头：Spring Security 支持将各种安全头添加到响应中。
- 匿名身份验证：允许未经过身份验证的用户访问。

除以上介绍的功能之外，还提供了很多功能，有具体需求的读者可以查看官方文档"对症下药"。笔者就不在这里做过多介绍了，毕竟 Spring Security 不是一节内容能介绍完的。

7.2.2 使用 Spring Security 做权限控制

接下来还是使用 7.1 节 Spring Boot 结合 Shiro 的场景，使用 Spring Boot 结合 Spring Security 实现同样的功能。

1. 场景及初始化数据

场景在这里不做过多介绍，可以查看 7.1 节的场景。数据库表也没有做大量修改，稍微修改了一下 Role 表的数据。需要注意，这里由原来的 user 修改成了 ROLE_USER，admin 修改成了 ROLE_ADMIN（稍后会进行解释）。初始化数据 SQL 如代码清单 7-17 所示。

代码清单 7-17 Spring Security 项目初始化数据代码

```
INSERT INTO `menu`(`menu_id`, `menu_name`) VALUES (1, '/add');
INSERT INTO `menu`(`menu_id`, `menu_name`) VALUES (2, '/delete');
INSERT INTO `menu`(`menu_id`, `menu_name`) VALUES (3, '/update');
INSERT INTO `menu`(`menu_id`, `menu_name`) VALUES (4, '/select');
INSERT INTO `role`(`role_id`, `role_name`) VALUES (1, 'ROLE_ADMIN');
INSERT INTO `role`(`role_id`, `role_name`) VALUES (2, 'ROLE_USER');
INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (1, 1);
INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (1, 2);
```

```
INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (1, 3);
INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (1, 4);
INSERT INTO `role_menu`(`role_id`, `menu_id`) VALUES (2, 4);
INSERT INTO `user`(`user_id`, `pass_word`, `user_name`) VALUES (1, '123',
'dalaoyang');
INSERT INTO `user`(`user_id`, `pass_word`, `user_name`) VALUES (2, '123',
'xiaoli');
INSERT INTO `user_role`(`role_id`, `user_id`) VALUES (1, 1);
INSERT INTO `user_role`(`role_id`, `user_id`) VALUES (2, 2);
```

2. 依赖文件及配置文件

新建项目，依赖内容与 7-1 小节类似，只需要将 Shiro 依赖替换为 Spring Security 依赖，完整内容如代码清单 7-18 所示。

代码清单 7-18　Spring Security 项目依赖文件代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>net.sourceforge.nekohtml</groupId>
    <artifactId>nekohtml</artifactId>
    <version>1.9.15</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

配置文件这里不再赘述，都是关于数据库和 JPA 的配置，如需查阅，可以在本书源代码处查看。

3. 实体类

实体类也可以采用 7.1 节的实体类，由于篇幅原因，代码就不再展示了。UserRepository 类同样提供了一个 findByUserName 方法。另外，由于场景需求，需要创建一个 RoleRepository，如代码清单 7-19 所示。

代码清单 7-19　Spring Security 项目 RoleRepository 类代码

```
public interface RoleRepository extends JpaRepository<Role,Integer> {
}
```

4. SecurityConfig

使用 Spring Security 进行安全管理需要使 SecurityConfig 继承 WebSecurityConfigurerAdapter 类，并且使用 HttpSecurity 的一切安全策略进行配置。本文使用的配置如下。

- authorizeRequests：配置一些资源或链接的权限认证。
- antMatchers：配置哪些资源或链接需要被认证。
- permitAll：设置完全允许访问的资源和链接。
- hasRole：配置需要认证的资源或链接的角色。需要注意，若这里需要配置权限为 USER，则用户需要拥有权限 ROLE_USER，这就是初始化脚本中权限内容修改的原因。
- formLogin：设置 form 表单提交配置。
- loginPage：设置一个自定义的登录页面 URL。
- failureUrl：设置一个自定义的登录失败的 URL。
- successForwardUrl：设置一个登录成功后自动跳转的 URL。
- accessDeniedPage：设置拒绝访问的 URL。
- logoutSuccessUrl：设置退出登录的 URL。

正如初始化脚本中可以看到的，在数据库中设置了两个用户，笔者在内存中使用 configureGlobal 设置了两个用户 test 和 admin，其中 test 用户的初始密码是 123，权限为 USER，admin 用户的初始密码 123，权限是 ADMIN 和 USER。由于本文没有给密码设置加密，因此需要定义一个 NoOpPasswordEncoder 的 Bean 来设置密码不加密。完整 SecurityConfig 内容如代码清单 7-20 所示。

代码清单 7-20　Spring Security 项目 SecurityConfig 类代码

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private RoleRepository roleRepository;
    @Autowired
    private MyUserDetailsService myUserDetailsService;
```

```
    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception{
        //配置资源文件，其中/css/**、/index 可以任意访问，/select 需要 USER 权限，
/delete 需要 ADMIN 权限
        httpSecurity
                .authorizeRequests()
                .antMatchers("/css/**", "/index").permitAll()
                .antMatchers("/select").hasRole("USER")
                .antMatchers("/delete").hasRole("ADMIN");
        //动态加载数据库中的角色权限
        List<Role> roleList = roleRepository.findAll();
        for(Role role : roleList){
            List<Menu> menuList = role.getMenuList();
            for (Menu menu : menuList){            //在 Spring Security 中校验权
限的时候，会自动在权限前面加 ROLE_，所以我们需要将数据库中配置的 ROLE_截取掉
                String roleName = role.getRoleName().replace("ROLE_","");
                String menuName = "/" + menu.getMenuName();
                httpSecurity
                        .authorizeRequests()
                        .antMatchers(menuName)
                        .hasRole(roleName);
            }
        }
        //配置登录请求/login 登录失败请求/login_error 登录成功请求/
        httpSecurity
                .formLogin()
                .loginPage("/login")
                .failureUrl("/login_error")
                .successForwardUrl("/");
        //登录异常，如权限不符合，请求/401
        httpSecurity
                .exceptionHandling().accessDeniedPage("/401");
        //注销登录，请求/logout
        httpSecurity
                .logout()
                .logoutSuccessUrl("/logout");
    }

    @Bean
    public static NoOpPasswordEncoder passwordEncoder() {
        return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
    }
```

```
        //根据用户名密码实现登录
        @Autowired
        public void configureGlobal(AuthenticationManagerBuilder
authenticationManagerBuilder) throws Exception {
            authenticationManagerBuilder
                    .inMemoryAuthentication()
                    //.passwordEncoder(new BCryptPasswordEncoder())
                    .withUser("test").password("123").roles("USER")
                    .and()
                    .withUser("admin").password("123").roles("ADMIN","USER");
            authenticationManagerBuilder.userDetailsService
(myUserDetailsService);
        }
    }
```

5. MyUserDetailsService

使用数据库认证用户需要自定义一个类来实现 UserDetailsService 重写 loadUserByUsername 方法进行认证授权，如代码清单 7-21 所示。

代码清单 7-21　Spring Security 项目 MyUserDetailsService 代码

```
    @Service
    public class MyUserDetailsService implements UserDetailsService {
        @Autowired
        private  UserRepository userRepository;

        @Override
        public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
            User user = userRepository.findByUserName(username);
            if (user == null){
                throw new UsernameNotFoundException("用户不存在！");
            }
            List<SimpleGrantedAuthority> simpleGrantedAuthorities = new
ArrayList<>();
            for (Role role : user.getRoleList()) {
                simpleGrantedAuthorities.add(new SimpleGrantedAuthority
(role.getRoleName()));
            }
            return new org.springframework.security.core.userdetails.User
(user.getUserName(), user.getPassWord(), simpleGrantedAuthorities);
        }
    }
```

6. 页面及 Controller

页面没有什么改变，基本上和 7.1 节一致，读者可以查看本书的源代码。接下来创建一个 TestController 进行页面跳转，如代码清单 7-22 所示。

代码清单 7-22　Spring Security 项目 TestController 类代码

```
@Controller
public class TestController {
    @RequestMapping({"/","/index"})
    public String index(){
        return"index";
    }

    @RequestMapping("/select")
    public String select(){
        return "select";
    }

    @RequestMapping("/delete")
    public String delete(){
        return "delete";
    }

    @RequestMapping("/login")
    public String login(){
        return "login";
    }

    @RequestMapping("/login_error")
    public String login_error(Model model){
        model.addAttribute("login_error", "用户名或密码错误");
        return "login";
    }

    @RequestMapping("/logout")
    public String logout(Model model){
        model.addAttribute("login_error", "注销成功");
        return "login";
    }

    @RequestMapping("/401")
    public String error(){
        return "401";
    }
}
```

7. 测试

推荐的测试方式与 7.1 节大致一致，这里不再赘述。

7.3 小　　结

本章学习了 Spring Boot 和 Apache Shiro 及 Spring Security 的整合，虽然整合得深度不高，但是已经将二者结合起来，可以在接下来的工作中有一定的基础。同时，如果要对安全框架进行扩展使用，也可以在本书的基础上使用。

第 8 章

Spring Boot 的监控之旅

监控是一个系统长期运营的必要保障，我们可以做一个这样的假设，当马路上不再有监控设备时，许多违章违纪的车辆将会钻法律的空子，不受法律的管理，长期这样，交通秩序将不再得到保障。而对于软件系统来说，监控同样必不可少，它可以在系统出现问题的时候自动提示系统维护人员，可以使出现的问题及时得到修复。本章笔者将带领大家学习 Spring Boot 常用的监控。

8.1　使用 actuator 监控

8.1.1　actuator 是什么

在 Spring Boot 的众多 Starter POMs 中有一个特殊的模块，不同于其他模块大多用于开发业务功能或连接一些其他外部资源，完全是一个用于暴露自身信息的模块，主要用于监控与管理，它就是 spring-boot-starter-actuator。

spring-boot-starter-actuator 模块的实现对于实施微服务的中小团队来说，可以有效地减少监控系统在采集应用指标时的开发量。当然，它并不是万能的，有时我们需要对其做一些简单的扩展来帮助我们实现自身系统个性化的监控需求。下面将详细介绍关于 spring-boot-starter-actuator 模块的内容，包括它原生提供的端点以及一些常用的扩展和配置方式。

8.1.2　如何使用 actuator

在 Spring Boot 中使用 actuator 很简单，只需要将项目加入 spring-boot-starter-actuator 依赖即可。这里为了方便观察，也加入了 spring-boot-starter-web 依赖，如代码清单 8-1 所示。

代码清单 8-1　Spring Boot-Actuator 项目依赖文件代码

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

8.1.3　actuator 监控介绍

没有特殊需求的话，其实到这里已经配置完成了，已经为应用程序开启了很多 actuator 端点。在 Spring Boot 应用中内置了很多端点，当然也支持添加自己需要的端点。例如，health 端点就提供了应用程序的健康信息。

大部分 actuator 端点都是可以独立进行的，通过对端点的启用和禁用可以控制是否创建端点用于查看信息。Spring Boot 应用通过 JMX 或者 HTTP 公开端点，大多数应用程序都是使用 HTTP 暴露端点的，端点使用前缀/actuator 加上端点 ID 来访问。例如，在默认情况下，health 端点映射到/actuator/health。

表 8-1 是 actuator 暴露的端点。

表 8-1　actuator 暴露的端点

HTTP 方法	ID	描述	默认情况下是否启用
GET	auditevents	显示应用程序的审核事件信息	是
GET	beans	显示应用程序中所有 Spring Bean 的完整列表	是
GET	conditions	显示在配置和自动配置类上评估的条件以及它们匹配或不匹配的原因	是
GET	configprops	显示配置列表，包括默认配置	是
GET	env	显示 Spring 的环境变量	是
GET	flyway	显示已应用的任何 Flyway 数据库迁移	是
GET	health	查看应用健康信息	是
GET	httptrace	显示 HTTP 跟踪信息(默认情况下显示最后 100 个)	是
GET	info	获取应用程序定制信息，这些信息提供 info 打头的属性	是
GET	loggers	显示和修改应用程序中记录器的配置	是
GET	liquibase	显示已应用的任何 Liquibase 数据库迁移	是
GET	metrics	显示当前应用程序的“指标”信息	是

（续表）

HTTP 方法	ID	描述	默认情况下是否启用
GET	mappings	显示所有@RequestMapping 路径的整理列表	是
GET	scheduledtasks	显示应用程序中的计划任务	是
GET	sessions	允许从 Spring Session 支持的会话存储中检索和删除用户会话。注：Spring Session 不支持 Web 响应式编程	是
POST	shutdown	允许应用程序正常关闭	是
GET	threaddump	执行线程转储	否
GET	heapdump	返回 GZip 压缩 hprof 堆转储文件	是
GET	jolokia	通过 HTTP 公开 JMX bean（当 Jolokia 在类路径上时，不适用于 WebFlux）	是
GET	logfile	返回日志文件的内容（如果已设置 logging.file 或 logging.path 属性）。支持使用 HTTP Range 标头来检索部分日志文件的内容	否
GET	prometheus	可以由 Prometheus 服务器抓取的格式公开指标	是

虽然大部分端点在默认情况下都是启用状态，但是在 Spring Boot 应用中，默认只开启 info 端点和 health 端点。其余端点都需要通过声明属性来开启，如代码清单 8-2 所示。

代码清单 8-2　开启全部端点相关代码

```
management.endpoints.web.exposure.include= *
```

通过以上设置可以开启所有默认启用的端点。当然，我们也可以根据 ID 指定开启端点，如代码清单 8-3 所示。

代码清单 8-3　开启局部端点相关代码

```
management.endpoints.web.exposure.include= heapdump,env
```

如果想要开启 shutdown 端点，那么可以使用如下配置使 shutdown 端点生效，如代码清单 8-4 所示。

代码清单 8-4　开启 shutdown 端点相关代码

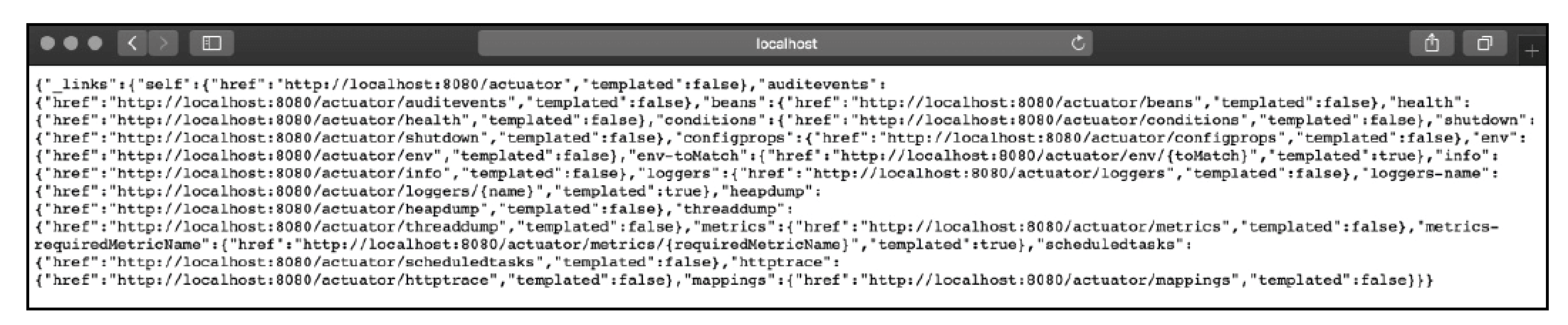
```
management.endpoint.shutdown.enabled = true
```

其实，在 Spring Boot 应用程序中，启动项目后在日志上就可以看到开启的 Web 端点，如图 8-1 所示。

还有一种查看方式，就是通过 HTTP 请求访问/actuator，如图 8-2 所示。

```
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/auditevents],methods=[GET],produces=[application/vnd.spring-boot.actua
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/beans],methods=[GET],produces=[application/vnd.spring-boot.actuator.v2
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/health],methods=[GET],produces=[application/vnd.spring-boot.actuator.v
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/conditions],methods=[GET],produces=[application/vnd.spring-boot.actuat
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/shutdown],methods=[POST],produces=[application/vnd.spring-boot.actuato
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/configprops],methods=[GET],produces=[application/vnd.spring-boot.actua
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/env],methods=[GET],produces=[application/vnd.spring-boot.actuator.v2+j
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/env/{toMatch}],methods=[GET],produces=[application/vnd.spring-boot.act
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/info],methods=[GET],produces=[application/vnd.spring-boot.actuator.v2+
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/loggers],methods=[GET],produces=[application/vnd.spring-boot.actuator.
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/loggers/{name}],methods=[GET],produces=[application/vnd.spring-boot.ac
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/loggers/{name}],methods=[POST],consumes=[application/vnd.spring-boot.a
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/heapdump],methods=[GET],produces=[application/octet-stream]}" onto pub
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/threaddump],methods=[GET],produces=[application/vnd.spring-boot.actuat
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/metrics],methods=[GET],produces=[application/vnd.spring-boot.actuator.
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/metrics/{requiredMetricName}],methods=[GET],produces=[application/vnd.
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/scheduledtasks],methods=[GET],produces=[application/vnd.spring-boot.ac
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/httptrace],methods=[GET],produces=[application/vnd.spring-boot.actuato
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/mappings],methods=[GET],produces=[application/vnd.spring-boot.actuator
INFO 1813 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator],methods=[GET],produces=[application/vnd.spring-boot.actuator.v2+json
```

图 8-1 Spring Boot-Actuator 项目通过启动日志查看 actuator 端点



{"_links":{"self":{"href":"http://localhost:8080/actuator","templated":false},"auditevents":
{"href":"http://localhost:8080/actuator/auditevents","templated":false},"beans":{"href":"http://localhost:8080/actuator/beans","templated":false},"health":
{"href":"http://localhost:8080/actuator/health","templated":false},"conditions":{"href":"http://localhost:8080/actuator/conditions","templated":false},"shutdown":
{"href":"http://localhost:8080/actuator/shutdown","templated":false},"configprops":{"href":"http://localhost:8080/actuator/configprops","templated":false},"env":
{"href":"http://localhost:8080/actuator/env","templated":false},"env-toMatch":{"href":"http://localhost:8080/actuator/env/{toMatch}","templated":true},"info":
{"href":"http://localhost:8080/actuator/info","templated":false},"loggers":{"href":"http://localhost:8080/actuator/loggers","templated":false},"loggers-name":
{"href":"http://localhost:8080/actuator/loggers/{name}","templated":true},"heapdump":
{"href":"http://localhost:8080/actuator/heapdump","templated":false},"threaddump":
{"href":"http://localhost:8080/actuator/threaddump","templated":false},"metrics":{"href":"http://localhost:8080/actuator/metrics","templated":false},"metrics-
requiredMetricName":{"href":"http://localhost:8080/actuator/metrics/{requiredMetricName}","templated":true},"scheduledtasks":
{"href":"http://localhost:8080/actuator/scheduledtasks","templated":false},"httptrace":
{"href":"http://localhost:8080/actuator/httptrace","templated":false},"mappings":{"href":"http://localhost:8080/actuator/mappings","templated":false}}}

图 8-2 Spring Boot-Actuator 项目通过 HTTP 请求查看 actuator 端点

8.1.4 保护 HTTP 端点

Spring Boot 应用程序提供的 actuator 端点虽然为我们提供了一定的便利，但若没有安全限制，则会有一定的风险，比如 shutdown 端点随意暴露的话，应用的启停就会被“坏人”利用。

这时我们可以像使用任何其他敏感 URL 一样注意保护 HTTP 端点。若存在 Spring Security，则默认使用 Spring Security 的内容协商策略来保护端点。例如，如果你希望为 HTTP 端点配置自定义安全性，只允许具有特定角色的用户访问它们，Spring Boot 提供了一些 RequestMatcher 可以与 Spring Security 结合使用的便捷对象。

在项目中加入 spring-boot-starter-security 依赖，如代码清单 8-5 所示。

代码清单 8-5 Spring Boot-Actuator 项目新增依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

当在应用程序中加入 spring-boot-starter-security 依赖后，再次在浏览器中访问 actuator 端点时，页面如图 8-3 所示。

因为我们没有为 Spring Security 设置用户密码，所以暂时无法登录。接下来，在配置文件中为 Spring Security 设置一个安全用户，如代码清单 8-6 所示。

代码清单 8-6 Spring Boot-Actuator 项目配置 Spring Security 安全用户

```
spring.security.user.name=admin
spring.security.user.password=123456
```



图 8-3　Spring Boot-Actuator 项目通过 HTTP 请求登录验证

重启项目，再次访问 actuator 端点时，输入正确的用户名和密码即可正常查看 actuator 端点的信息，与图 8-2 中的内容一样。

actuator 也支持如 7-2 小节那样，通过配置权限来决定哪些方法可以被符合权限的用户访问，新增 ActuatorSecurity 配置类，继承 WebSecurityConfigurerAdapter 并且重写了 configure()方法，如代码清单 8-7 所示。

代码清单 8-7　Spring Boot-Actuator 项目 ActuatorSecurity 类代码

```
@Configuration
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatcher(EndpointRequest.toAnyEndpoint()).
authorizeRequests()
                .anyRequest().hasRole("ENDPOINT_ADMIN")
                .and()
                .httpBasic();
    }
}
```

在 ActuatorSecurity 配置类中设置任意 actuator 端点可以被安全用户登录，但是安全用户需要具备 ENDPOINT_ADMIN 权限，如果安全用户没有权限，访问就会报 403 错误（权限不足），如图 8-4 所示。

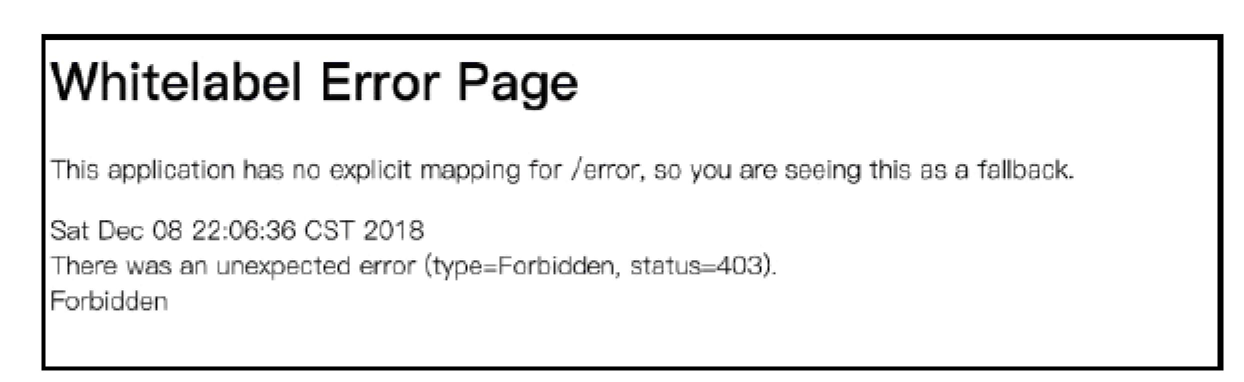


图 8-4　Spring Boot-Actuator 项目 403 错误页面

接下来，在配置文件中为安全用户赋予 ENDPOINT_ADMIN 权限，如代码清单 8-8 所示。

代码清单 8-8　Spring Boot-Actuator 项目将安全用户赋予权限

```
spring.security.user.roles=ENDPOINT_ADMIN
```

重启项目，再次访问可以正常查看 actuator 端点信息。

当然，我们也可以设置无须身份验证即可访问所有执行器端点。将 ActuatorSecurity 配置类的 @Configiration 注释上，其实就是取消 ActuatorSecurity 配置，新建 ActuatorNoSecurity 配置类。这里配置所有用户可以访问 actuator 端点，如代码清单 8-9 所示。

代码清单 8-9　Spring Boot-Actuator 项目无须安全验证配置类

```
    @Configuration
    public class ActuatorNoSecurity extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http.requestMatcher(EndpointRequest.toAnyEndpoint()).
authorizeRequests()
                    .anyRequest().permitAll();
        }
    }
```

重启项目后，即使加入了 Spring Security，也无须身份验证就可以访问 actuator 端点。

8.1.5　健康信息

health 端点是查看 Spring Boot 应用程序健康状况的端点，如果没有特殊设置，显示的信息就比较少，如下所示：

```
{"status":"UP"}
```

我们可以通过在配置文件中设置 management.endpoint.health.show-details 来决定 health 端点的细节信息是否展示。以下为 health 端点的细节属性。

- never：细节信息详情永远都不展示。
- when-authorized：细节详情只对授权用户显示。
- always：细节详情显示给所有用户。

属性默认值为 never，当存在授权用户时，如果一个用户处于一个或者多个端点的角色，则将被视为已经获得授权。如果端点没有配置角色，则认为所有经过身份验证的用户都已获得授权。我们可以使用 management.endpoint.health.roles 属性配置角色。

这里以 always 为例，在配置文件中加入配置，如代码清单 8-10 所示。

代码清单 8-10　Spring Boot-Actuator 项目加入配置

```
management.endpoint.health.show-details=always
```

重启项目，重新访问 http://localhost:8080/actuator/health，这时健康端点信息如下：

```
{"status":"UP","details":{"diskSpace":{"status":"UP","details":{"total":2
50790436864,"free":101807063040,"threshold":10485760}}}}
```

当然，详情开放不只是针对 health 端点，其他端点同样适用。

在这个测试项目中，可能看不到 health 端点的作用，因为这个项目中没有配置其他相关的信息。其实健康信息的内容是从 HealthIndicators 中收集应用程序中定义的所有 bean 中的上下文信息，其中包含一些自动配置的 HealthIndicators，也可以编写自己的健康信息 bean。Spring Boot 默认会自动配置以下 HealthIndicators。

- CassandraHealthIndicator：检查 Cassandra 数据库是否已启动。
- DiskSpaceHealthIndicator：检查磁盘空间是否不足。
- DataSourceHealthIndicator：检查是否可以获得连接的 DataSource。
- ElasticsearchHealthIndicator：检查 Elasticsearch 集群是否已启动。
- InfluxDbHealthIndicator：检查 InfluxDB 服务器是否已启动。
- JmsHealthIndicator：检查 JMS 代理是否已启动。
- MailHealthIndicator：检查邮件服务器是否已启动。
- MongoHealthIndicator：检查 Mongo 数据库是否已启动。
- Neo4jHealthIndicator：检查 Neo4j 数据库是否已启动。
- RabbitHealthIndicator：检查 Rabbit 服务器是否已启动。
- RedisHealthIndicator：检查 Redis 服务器是否已启动。
- SolrHealthIndicator：检查 Solr 服务器是否已启动。

如果不想在项目中使用这些安全检查，就可以使用 management.health.defaults.enabled 属性来禁用它们，比如要禁用 Rabbit 安全检查，可以做如下设置，如代码清单 8-11 所示。

代码清单 8-11　Spring Boot-Actuator 项目禁用 Rabbit 服务器安全检查

```
management.health.rabbit.enabled=false
```

之前提到了自定义 HealthIndicators，接下来带领大家编写一个自定义的 HealthIndicators。其实要提供自定义健康状况信息，只需要编写一个实现 HealthIndicator 的类，重写 health 方法即可。这里编写一个简单的自定义 HealthIndicators，如代码清单 8-12 所示。

代码清单 8-12　Spring Boot-Actuator 项目自定义 HealthIndicators

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealthIndicator implements HealthIndicator {

    private final String defaultServerPort = "80";

    @Value("${server.port}")
    private String serverPort;

    @Override
    public Health health() {
```

```
            int errorCode = check();
            if (errorCode != 0) {
                return Health.down().withDetail("Error Code:", errorCode).build();
            }
            return Health.up().build();
        }

        public int check(){
            if(!defaultServerPort.equals(serverPort)){
                return 500;
            }
            return 0;
        }

    }
```

在 MyHealthIndicator 类中，其实只是做了一个检查，这里设置了一个默认的端口号。如果当前应用程序端口号不是默认端口号（80），就返回错误码 500；如果当前应用程序端口号为 8080，就启动项目，访问 http://localhost:8080/actuator/health，如下所示：


```
    {"status":"DOWN","details":{"my":{"status":"DOWN","details":{"Error
Code:":500}},"diskSpace":{"status":"UP","details":{"total":250790436864,"free
":101805297664,"threshold":10485760}}}}
```


在输出信息中可以看到我们自定义的信息已经打印出来了。

8.1.6　自定义应用程序信息

在 actuator 端点中可以公开自定义信息，比如在配置文件中设置 info.*，如代码清单 8-13 所示。

代码清单 8-13　Spring Boot-Actuator 项目自定义 info 属性暴露

```
info.encoding = UTF-8
info.jdk.version = 1.8
```

重启项目，访问 http://localhost:8080/actuator/info 后即可看到，这里不再展示。

8.1.7　自定义管理端点路径

之前介绍了，actuator 端点在使用 HTTP 访问时需要使用前缀/actuator，其实这个前缀也可以根据需求自定义修改，只需要在配置文件中配置 management.endpoints.web.base-path 属性即可，如代码清单 8-14 所示。

代码清单 8-14　Spring Boot-Actuator 项目自定义管理端点路径

```
management.endpoints.web.base-path=/manage
```

修改后，再访问 health 端点，就由/actuator/health 变成了/manage/health。

actuator 是 Spring Boot 的重要特性，关于 actuator 的内容还有很多，提供端点能够做到的监控也有很多，具体可以查看官网对于 actuator 端点的介绍（Spring Boot 2.0.3 版本的官网地址：https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/actuator-api//html/）。

8.2　使用 Admin 监控

8.2.1　什么是 Spring Boot Admin

Spring Boot Admin 是 Spring Boot 项目的一个社区项目，主要用于管理和监控 Spring Boot 应用程序。通常来说，应用程序向我们的 Spring Boot Admin Server（通过 HTTP）直接注册信息或者 Spring Boot Admin Server 通过使用 Spring Cloud（例如 Eureka、Consul）服务发现收集 Client 信息。UI 只是 Spring Boot Actuator 端点上的一个 AngularJs 应用程序（2.x 版本后使用 Vue）。

8.2.2　设置 Spring Boot Admin Server

Spring Boot Admin 应用分为 Spring Boot Admin Server 应用和 Spring Boot Admin Client 应用。其中，Spring Boot Admin Server 应用用于收集 Spring Boot Admin Client 应用的信息并对其进行监控等。接下来，我们创建一个 Spring Boot Admin Server 应用。

创建 Spring Boot Admin Server 应用的过程大致分为两步，首先创建一个 Spring Boot 应用程序，在 pom 文件中加入依赖，如代码清单 8-15 所示。

代码清单 8-15　Spring Boot-Admin-Server 项目依赖文件代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

然后在 Spring Boot 应用程序启动类中加入@EnableAdminServer，引入 Spring Boot Admin Server 配置，如代码清单 8-16 所示。

代码清单 8-16　Spring Boot-Admin-Server 项目依赖文件代码

```
@Spring BootApplication
@EnableAdminServer
public class Chapter83Application {
```

```
    public static void main(String[] args) {
        SpringApplication.run(Chapter83Application.class, args);
    }
}
```

到这里，Spring Boot-Admin-Server 项目就已经配置完成了。启动项目可以看到如图 8-5 所示的页面。

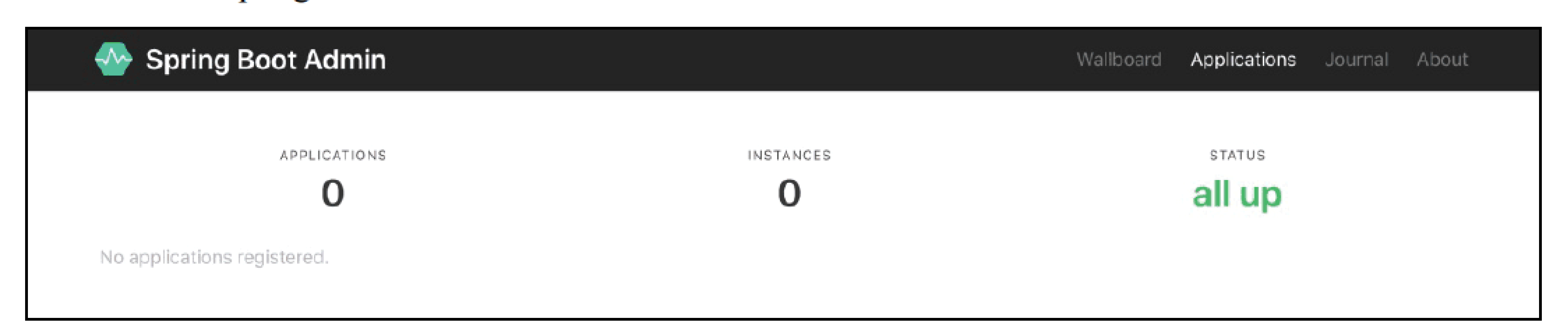


图 8-5　Spring Boot-Admin-Server 监控页面

由于还没有 Spring Boot Admin Client 应用注册到 Spring Boot-Admin-Server，因此现在实例数量还是 0。

8.2.3　Spring Cloud Eureka

可能到这里有人会问，这是一本讲解 Spring Boot 的书，为什么突然讲到 Spring Cloud 的组件 Eureka。因为 Spring Boot Admin Client 有一种方式是基于 Eureka 获取信息，所以这里介绍一下 Spring Cloud 的注册中心组件 Eureka。

引用官方的介绍：Eureka 是 Netflix 开发的服务发现框架，本身是一个基于 REST 的服务，主要用于定位运行在 AWS 域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的。Spring Cloud 将它集成在子项目 spring-cloud-netflix 中，以实现 Spring Cloud 的服务发现功能。

通俗地理解，Eureka 就是一个注册中心。本小节仅对 Eureka Server 进行简单的介绍，感兴趣的读者可以去 Spring Cloud 官网（官网地址：https://cloud.spring.io/spring-cloud-static/Finchley.SR2/single/spring-cloud.html）查看更多关于 Eureka 的信息。

接下来，笔者带领大家创建一个 Eureka Server。新建项目，在 pom 文件中加入 spring-cloud-starter-netflix-eureka-server 依赖（使用 Spring Cloud Finchley.SR1 版本）。完整 pom 文件内容如代码清单 8-17 所示。

代码清单 8-17　Spring Cloud-Eureka-Server 项目依赖文件代码

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.springboot</groupId>
    <artifactId>chapter8-2</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>chapter8-2</name>
    <description>chapter8-2</description>

    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>Finchley.SR1</spring-cloud.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server
</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

    <repositories>
        <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>

</project>
```

然后在启动类中加入注解@EnableEurekaServer，表明当前 Spring Boot 应用程序是一个 Eureka Server 程序，如代码清单 8-18 所示。

代码清单 8-18　Spring Cloud-Eureka-Server 项目启动类代码

```
@SpringBootApplication
@EnableEurekaServer
public class Chapter82Application {

    public static void main(String[] args) {
        SpringApplication.run(Chapter82Application.class, args);
    }
}
```

最后，我们只需要在配置文件中对 Eureka Server 应用程序进行配置，因为这里使用单节点 Eureka Server 应用，注意设置禁止向自己注册服务。配置文件如代码清单 8-19 所示。

代码清单 8-19　Spring Cloud-Eureka-Server 配置文件代码

```
server.port=8761
eureka.instance.hostname=localhost
eureka.client.service-url.defaultZone=http://${eureka.instance.hostname}:
${server.port}/eureka/

##禁止向自己注册
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

到这里，Eureka Server 应用配置完成了。启动项目，在浏览器中访问 http://localhost:8761，可以看到如图 8-6 所示的页面。

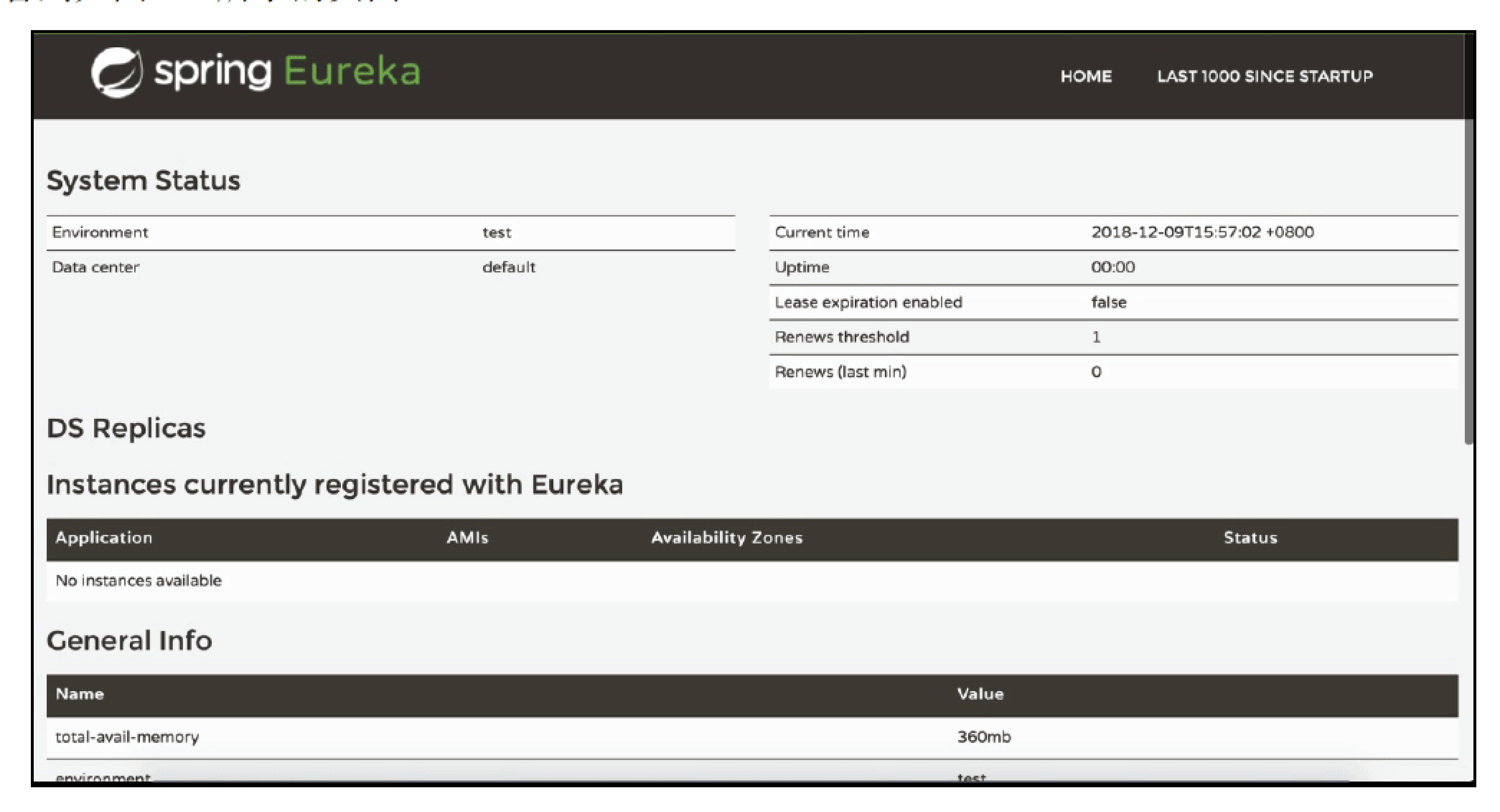


图 8-6　Spring Boot-Eureka Server 监控页面

与 Spring Boot Admin Server 一致，目前还没有实例注册进来，所以现在实例是空的，稍后会继续使用。

8.2.4　Spring Boot Admin Client 的使用

前面介绍了，使用 Spring Boot Admin Server 监控管理有两种方式，基于 Spring Cloud Discovery（Eureka 服务发现）或者对实例应用进行配置。接下来笔者带领大家分别对两种模式进行学习。

1. Spring Boot Admin 客户端

使用这种方式，所有需要使用 Spring Boot Admin Server 监控管理的应用程序都需要引入 spring-boot-admin-starter-client 依赖，如代码清单 8-20 所示。

代码清单 8-20　Spring Boot Admin Client 项目依赖文件代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.0.3</version>
</dependency>
```

接下来，在配置文件中配置 Spring Boot Admin Server 的地址，并且配置应用名称，以便在 Spring Boot Admin Server 页面查看。这里将端口号设置为 8081，如代码清单 8-21 所示。

代码清单 8-21　Spring Boot Admin Client 项目依赖文件代码

```
spring.boot.admin.client.url=http://localhost:8080
server.port=8081
spring.application.name=springboot-admin-client
```

启动 Spring Boot Admin Client 应用程序后，再来查看一下 Spring Boot Admin Server 监控页面，如图 8-7 所示。

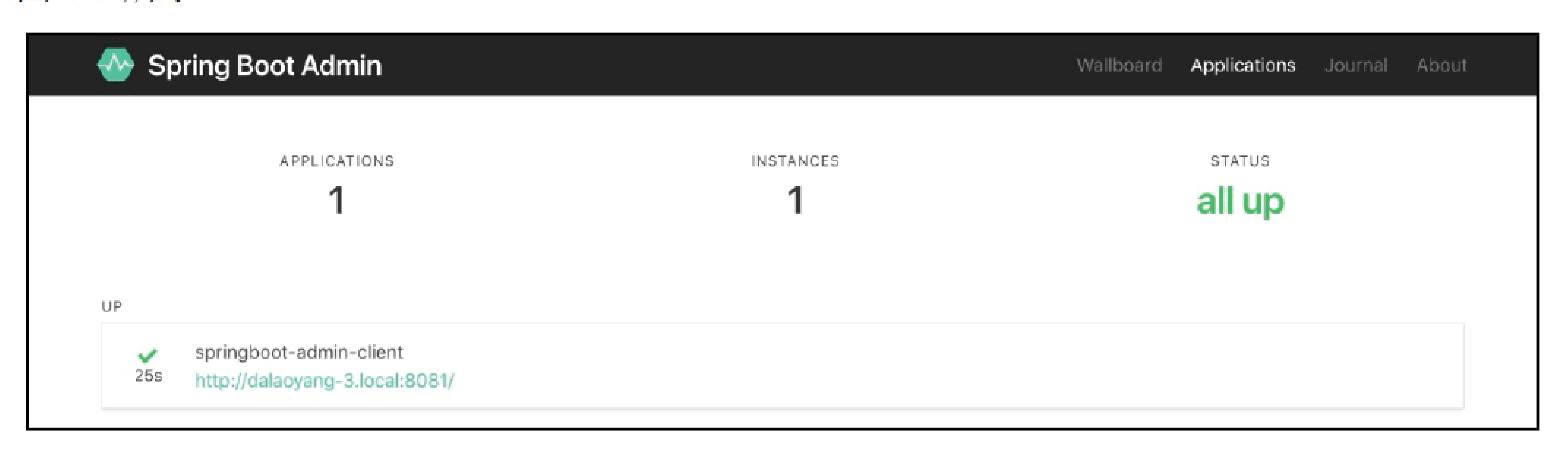


图 8-7　Spring Boot-Admin Server 监控页面

从图 8-7 中可以看到我们刚刚创建的应用实例已经注册到了 Spring Boot Admin Server 中，在 Spring Boot Admin Server 中会检测 Spring Boot Admin Client 实例的健康状况，其实就是检测 actuator 端点的 health 端点，因为当前应用实例状态为 up，所以状态显示为 all up。如果当前有任何应用不是 up 状态，就会显示状态为 down。

接下来单击实例，可以查看实例的详细信息，如图 8-8 所示。

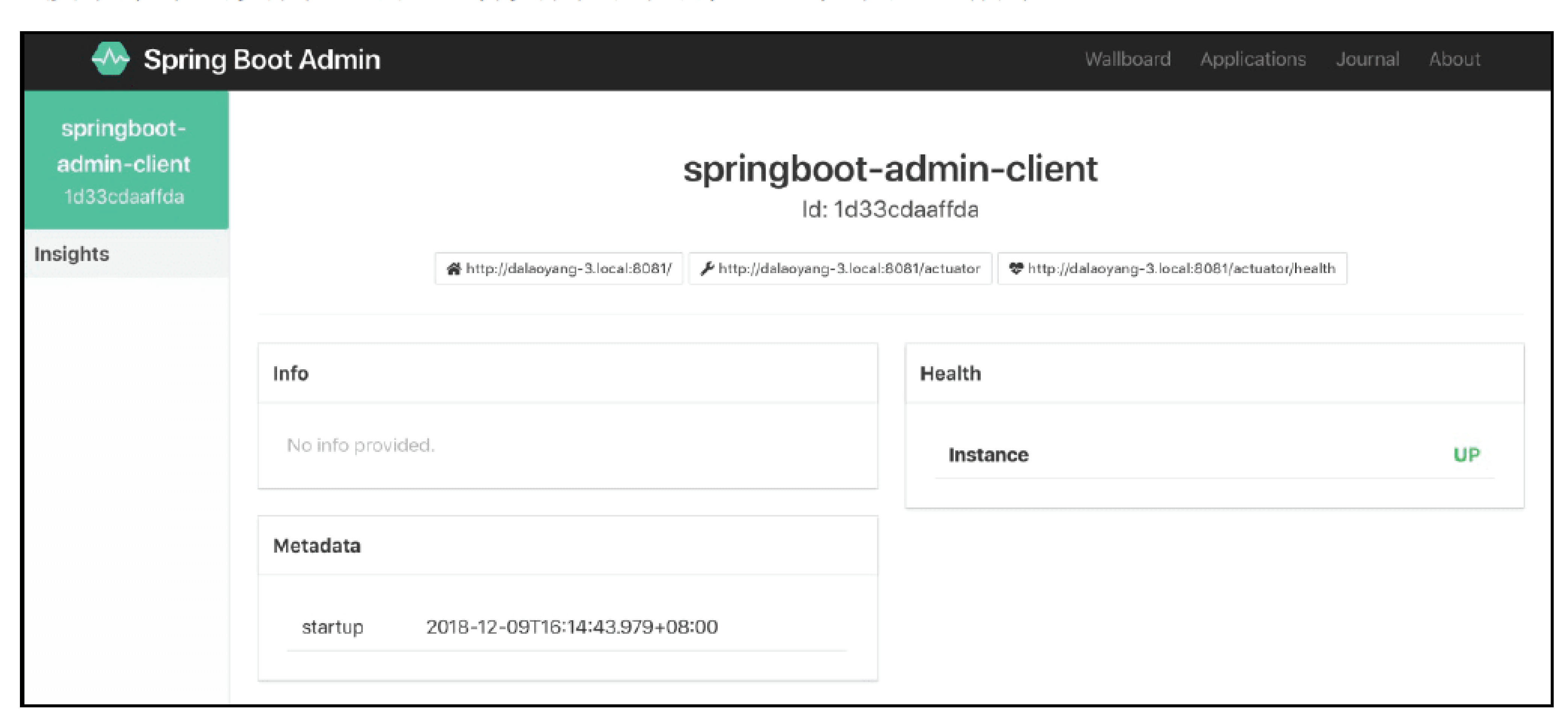


图 8-8　实例的详细信息

由于当前 Spring Boot Admin Client 应用程序没有为 actuator 开放更多的端点，因此这里仅能看到很少的信息。接下来我们为当前应用程序开放全部端点并且设置显示全部详细信息，然后查看 Spring Boot Admin Server 监控页面，如图 8-9 所示。

从图 8-9 中可以看到 Spring Boot-Admin Server 监控页面的数据随着开放的端点变多而变多，其实 Spring Boot-Admin Server 监控就是对 Spring Boot 应用程序的 actuator 端点进行一定的监控管理。

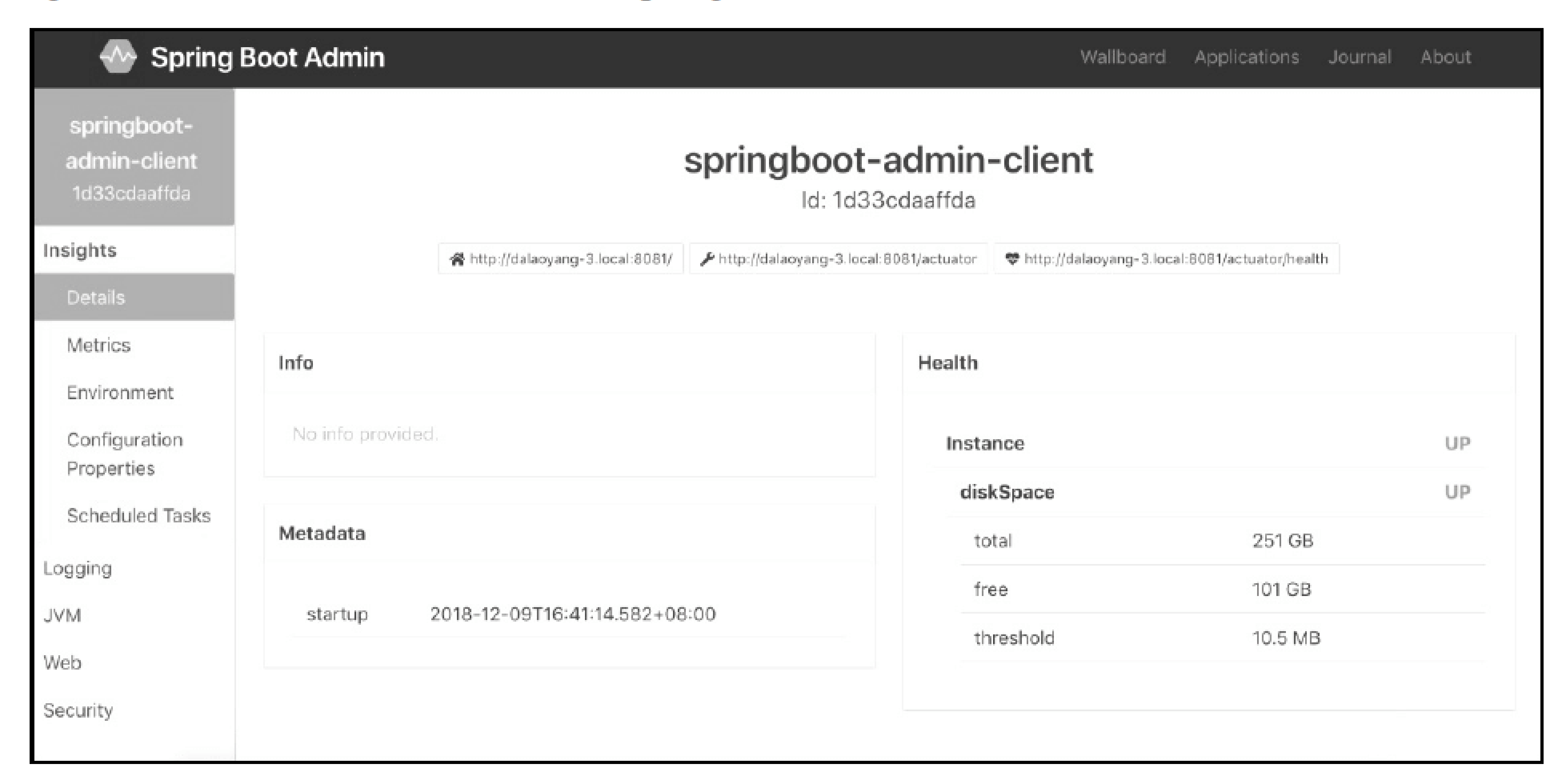


图 8-9　再次查看 Spring Boot-Admin Server 监控页面

2. 基于服务发现

接下来，我们使用 Eureka 的方式使用 Spring Boot-Admin Server 监控管理。新建项目，在项目中加入 spring-cloud-starter-netflix-eureka-client 依赖，如代码清单 8-22 所示。

代码清单 8-22　Spring Boot Eureka Client 项目依赖文件完整代码

```
    <?xml version="1.0" encoding="UTF-8"?>
    <project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>2.0.3.RELEASE</version>
            <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.springboot</groupId>
        <artifactId>chapter8-5</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>jar</packaging>
        <name>chapter8-5</name>
        <description>chapter8-5</description>

        <properties>
            <java.version>1.8</java.version>
```

```xml
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client
</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Finchley.SR1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
```

```
        </plugins>
    </build>

    <repositories>
        <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
</project>
```

在配置文件中配置 Eureka Server 的地址，并且开启全部 actuator 端点，设置端口号为 8082，如代码清单 8-23 所示。

代码清单 8-23　Spring Boot Admin Client 项目依赖文件代码

```
server.port=8082
spring.application.name=springboot-admin-client2
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
## eureka server 地址
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

然后将 Spring Boot Admin Server 应用程序加入 Eureka Client 依赖及配置，过程同上。全部配置完成后重启项目，先查看一下 Eureka Server 监控页面，如图 8-10 所示。

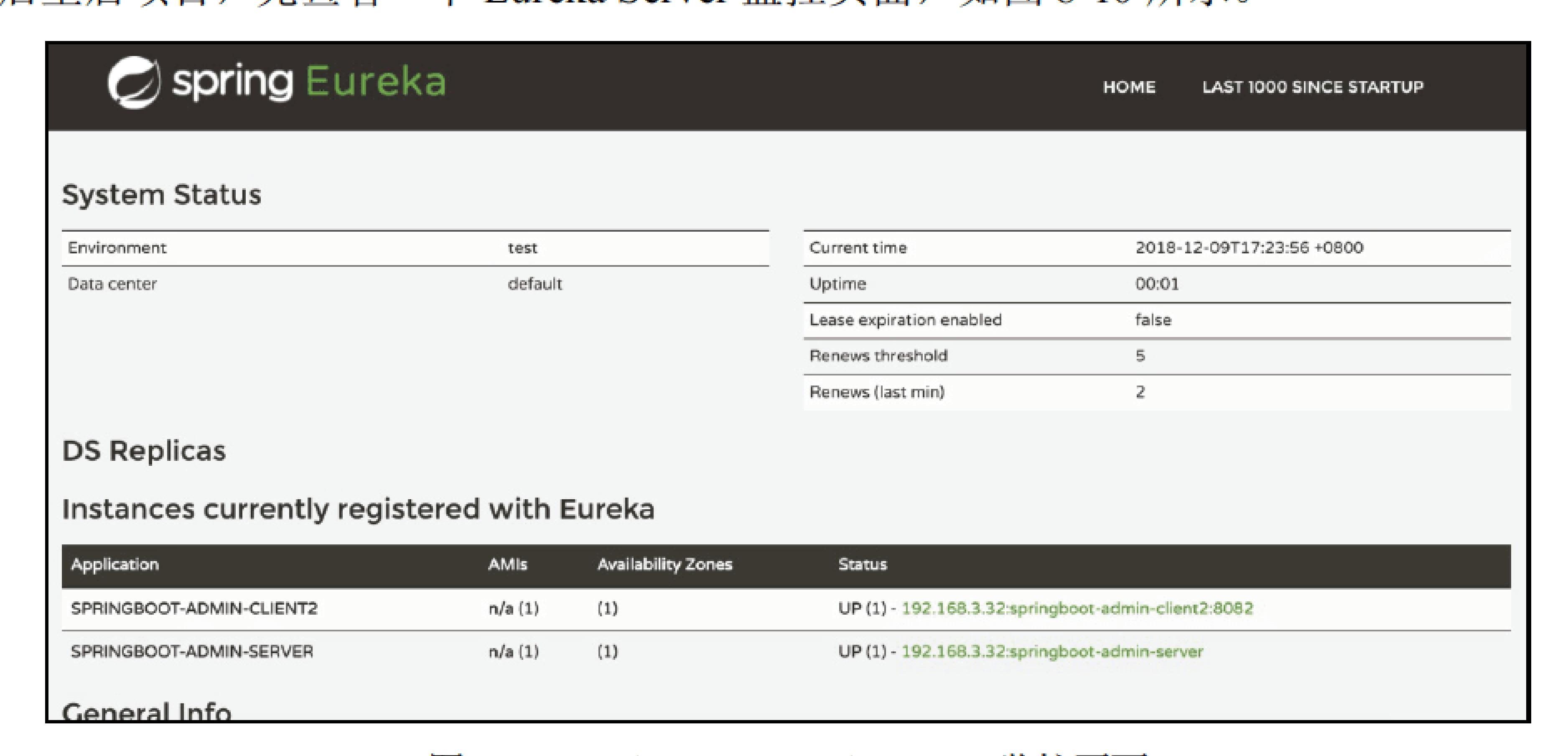


图 8-10　Spring Boot-Eureka Server 监控页面

可以看到实例部分已经存在两个使用 Eureka 的应用实例。接着查看 Spring Boot Admin Server 监控页面，如图 8-11 所示。

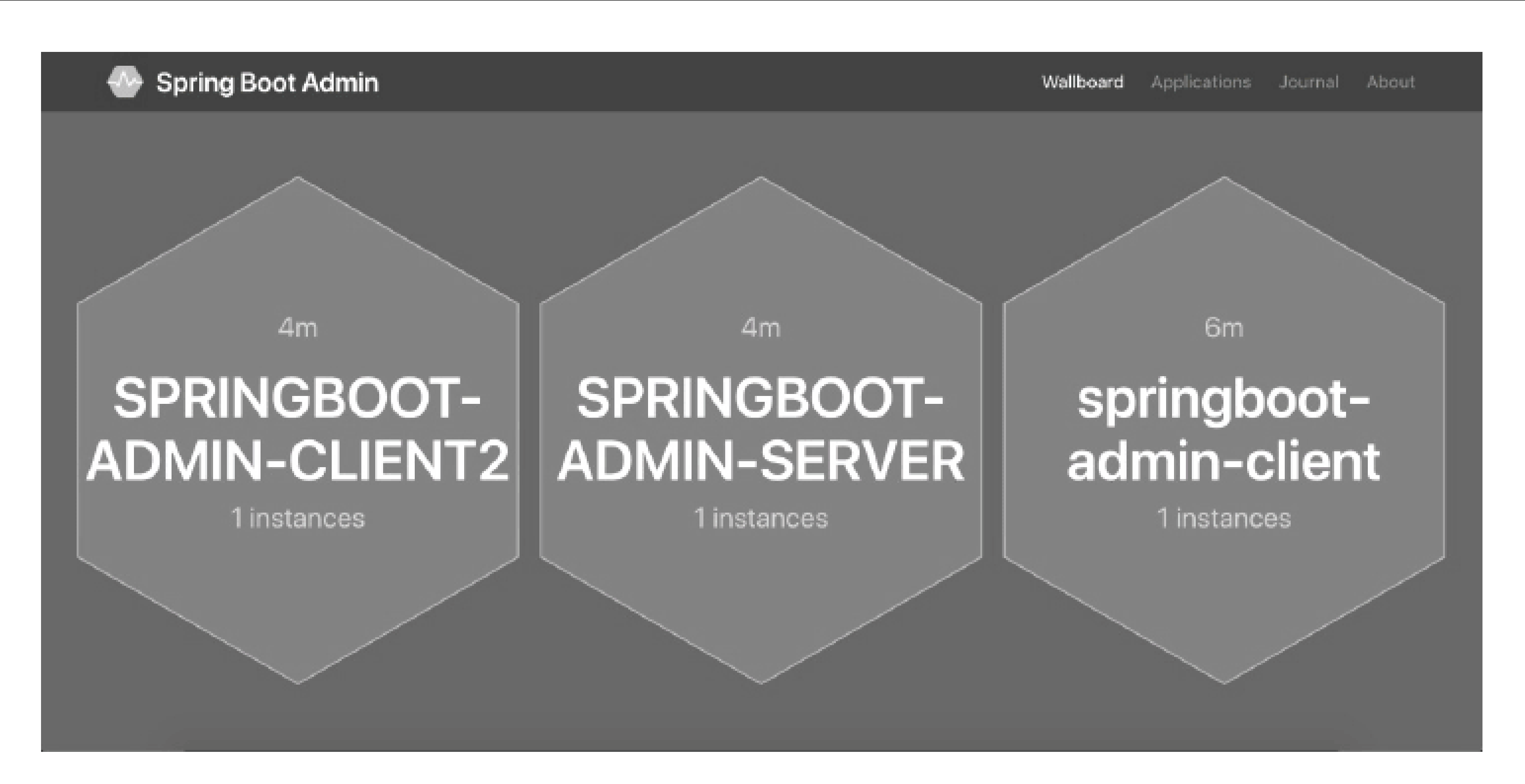


图 8-11 Spring Boot-Admin Server 监控页面

笔者分别介绍了使用 Spring Boot Admin 客户端和基于服务发现两种方式进行注册实例，都可以达到使用 Spring Boot Admin 监控的目的，这里笔者建议使用基于 Eureka 的方式，这样会更简单一些，无须基于每个应用频繁地配置。

8.2.5 安全验证

如果 Spring Boot Admin Server 监控页面可以随意查看，似乎不太安全。接下来笔者带领大家为 Spring Boot Admin Server 监控增加安全管理，在 pom 文件中加入 spring-boot-starter-security 依赖配置，如代码清单 8-24 所示。

代码清单 8-24 Spring Boot Admin Server 添加 Security 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

接下来加入一个配置类，使 actuator 端点可访问（这里设置登录用户可以查看全部端点），如代码清单 8-25 所示。

代码清单 8-25 Spring Boot Admin Server 添加 SecurityPermitAllConfig 配置代码

```
@Configuration
public class SecurityPermitAllConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().permitAll()
                .and().csrf().disable();
    }
}
```

在配置文件中配置用户名 admin、密码 123456，然后重启项目，访问 http://localhost:8080，Spring Boot Admin Server 为我们提供了友好的登录页面，如图 8-12 所示。

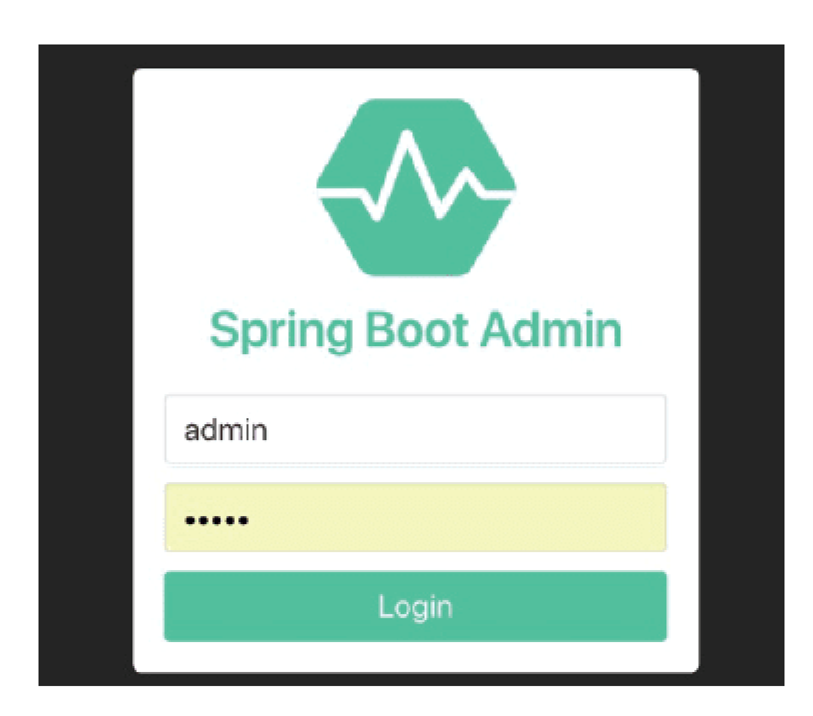


图 8-12　Spring Boot-Eureka Server 监控登录页面

使用用户名、密码登录后，就可以和之前一样查看 Spring Boot Admin Server 安全管理页面。

8.2.6　JMX-bean 管理

如果在 Spring Boot Admin Server 安全管理页面需要与 JMX-bean 交互，那么在应用程序中必须包含 Jolokia。由于 Jolokia 是基于 servlet 的，因此不支持响应式应用程序，WebFlux 在这里暂时不支持。使用 Jolokia 只需要引入 Jolokia 依赖，如代码清单 8-26。

代码清单 8-26　Jolokia 依赖代码

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
</dependency>
```

8.2.7　通知

Spring Boot Admin Server 不但提供了管理页面供我们查看，而且提供了一些通知，比如邮件通知、PagerDuty 通知、OpsGenie 通知等。本小节仅对邮件通知进行介绍。

邮件通知将作为使用 Thymeleaf 模板呈现的 HTML 电子邮件传递。如果需要启用邮件通知，就需要在项目中加入 spring-boot-starter-mail 依赖，如代码清单 8-27 所示。

代码清单 8-27　spring-boot-starter-mail 依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

在配置文件中新增邮箱相关配置信息，在后面的章节会介绍 Spring Boot 邮件发送的相关内容，这里不做过多解释。关于 Spirng Boot Admin 需要设置发送邮件地址和收件地址，如代码清单 8-28 所示。

代码清单 8-28　Spring Boot Admin Server 项目新增邮箱发送模板代码

```
spring.mail.host=smtp.qq.com
spring.mail.username=yangyang@dalaoyang.cn
spring.mail.password=邮件密码
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true
spring.boot.admin.notify.mail.from=yangyang@dalaoyang.cn
spring.boot.admin.notify.mail.to=yangyang@dalaoyang.cn
```

重启项目后查看邮件，可以看到如图 8-13 所示的页面。

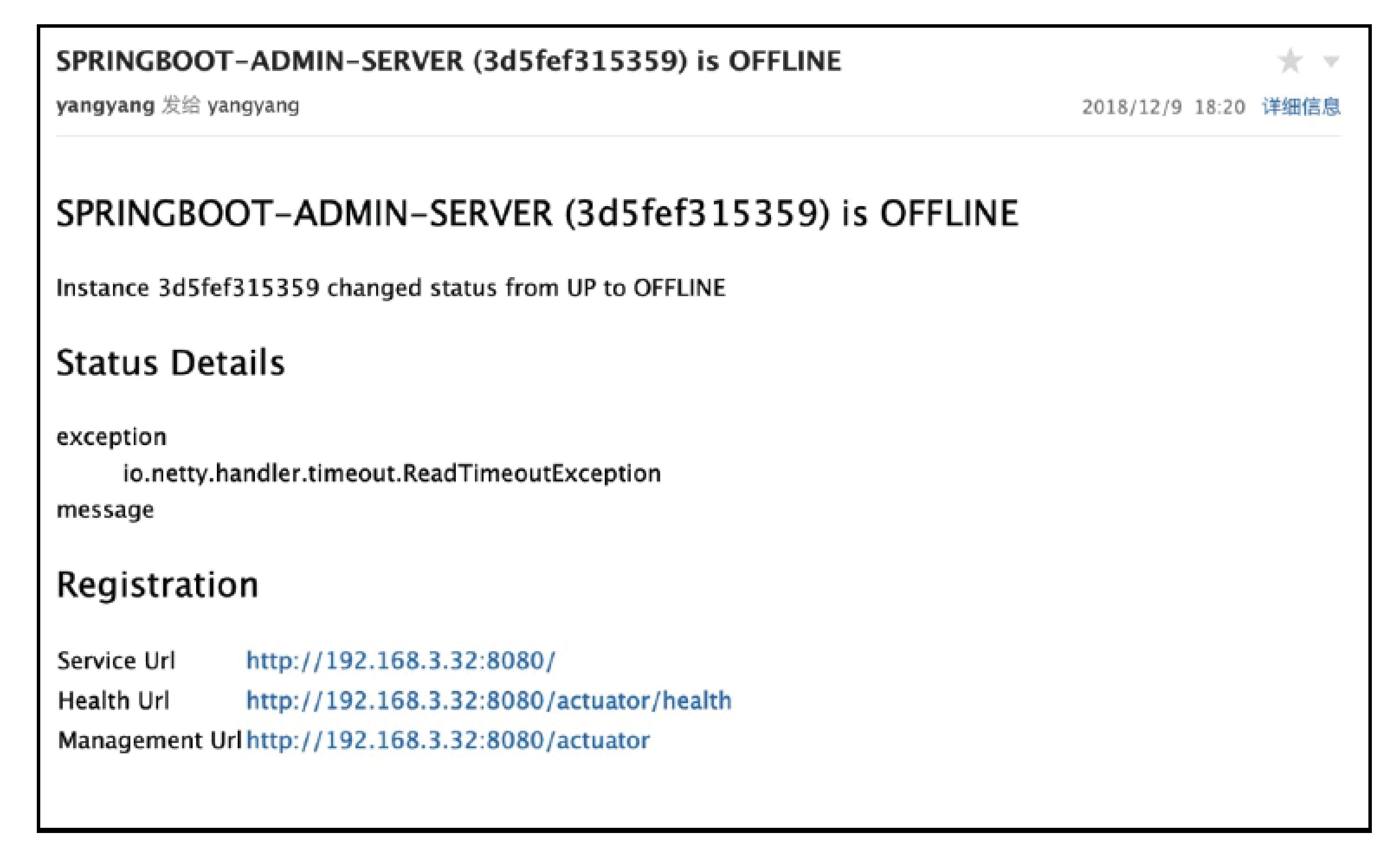


图 8-13　Spring Boot-Eureka Server 监控邮件发送

从图 8-13 中可以看到关于实例应用的状态变化、异常信息等，但是模板是英文的，如果需要自定义页面模板，那么可以新建一个 HTML，并且在配置文件中配置 spring.boot.admin.notify.mail.template。比如在 src/main/resources/文件夹下创建 status-changed.html 文件，然后在配置文件中配置 spring.boot.admin.notify.mail.template=classpath:/status-changed.html。以下是笔者根据官方提供的模板页面稍作修改的中文版，如代码清单 8-29 所示。

代码清单 8-29　Spring Boot Admin Server 项目新增邮箱相关配置代码

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <style>
        h1, h2, h3, h4, h5, h6 {
            font-weight: 400
        }
```

```
          ul {
             list-style: none
          }

          html {
             box-sizing: border-box
          }

          *, :after, :before {
             box-sizing: inherit
          }

          table {
             border-collapse: collapse;
             border-spacing: 0
          }

          td, th {
             text-align: left
          }

          body, button {
             font-family: BlinkMacSystemFont, -apple-system, "Segoe UI",
Roboto, Oxygen, Ubuntu, Cantarell, "Fira Sans", "Droid Sans", "Helvetica Neue",
Helvetica, Arial, sans-serif
          }

          code, pre {
             -moz-osx-font-smoothing: auto;
             -webkit-font-smoothing: auto;
             font-family: monospace
          }
       </style>
   </head>
   <body>
   <th:block th:remove="all">
       <!-- This block will not appear in the body and is used for the subject
-->
       <th:block th:remove="tag" th:fragment="subject">

          服务提醒：[[${instance.registration.name}]] (实例
ID:[[(${instance.id})]]) 状态变成了
          [[${event.statusInfo.status}]]
       </th:block>
```

```
    </th:block>
    <h1>服务名:<span th:text="${instance.registration.name}"/> (实例 ID:<span
th:text="${instance.id}"/>)
        状态变成了
        [[${event.statusInfo.status}]]
    </h1>
    <p>
        实例 <a th:if="${baseUrl}" th:href="@{${baseUrl + '/#/instances/' +
instance.id + '/'}}"><span
            th:text="${instance.id}"/></a>
        <span th:unless="${baseUrl}" th:text="${instance.id}"/>
        状态由 <span th:text="${lastStatus}"/> 变为 <span
th:text="${event.statusInfo.status}"/>
    </p>

    <h2>状态细节</h2>
    <dl th:fragment="statusDetails" th:with="details = ${details ?:
event.statusInfo.details}">
        <th:block th:each="detail : ${details}">
            <dt th:text="${detail.key}"/>
            <dd th:unless="${detail.value instanceof T(java.util.Map)}"
th:text="${detail.value}"/>
            <dd th:if="${detail.value instanceof T(java.util.Map)}">
                <dl th:replace="${#execInfo.templateName} :: statusDetails
(details = ${detail.value})"/>
            </dd>
        </th:block>
    </dl>

    <h2>注册信息</h2>
    <table>
        <tr th:if="${instance.registration.serviceUrl}">
            <td>服务地址</td>
            <td>
                <a th:href="${instance.registration.serviceUrl}"
th:text="${instance.registration.serviceUrl}"></a>
            </td>
        </tr>
        <tr>
            <td>健康地址</td>
            <td>
                <a th:href="${instance.registration.healthUrl}"
th:text="${instance.registration.healthUrl}"></a>
            </td>
```

```
        </tr>
        <tr th:if="${instance.registration.managementUrl}">
            <td>管理地址</td>
            <td>
                <a th:href="${instance.registration.managementUrl}"
th:text="${instance.registration.managementUrl}"></a>
            </td>
        </tr>
    </table>
    </body>
    </html>
```

使用自定义模板后，页面如图 8-14 所示。



图 8-14　Spring Boot-Eureka Server 监控邮件发送

笔者只是将对应英文修改成了中文，没有做过多修改，具体使用可以仔细考量需要得到的信息。

8.3　Prometheus+Grafana 监控

前面介绍了 Spring Boot Admin 监控 Spring Boot 应用程序，接下来将介绍由 Prometheus 结合 Grafana 搭建 Spring Boot 监控平台。

8.3.1　Prometheus 的安装

Prometheus（官网地址：https://prometheus.io/）是一个开源的系统监控和报警的工具包，最初由 SoundCloud 发布。Prometheus 通过在这些目标上抓取指标 HTTP 端点来收集受监控目标的指标。

由于 Prometheus 以同样的方式公开数据，因此它也可以掠夺和监控自身的健康状况，感兴趣的读者可以查阅官方文档。

以 Linux 系统为例进行介绍，到 Prometheus 官方下载页（官网下载页地址：https://prometheus.io/download/）下载安装包，如代码清单 8-30 所示。

代码清单 8-30　下载 Prometheus 代码

```
wget https://github.com/prometheus/prometheus/releases/download/v2.6.1/
prometheus-2.6.1.linux-amd64.tar.gz
```

下载完成后，解压文件，如代码清单 8-31 所示。

代码清单 8-31　解压 Prometheus 代码

```
tar xvfz prometheus-*.tar.gz
```

8.3.2　Grafana 的安装

Grafana（官网地址：https://grafana.com/）是一个深度分析的可视化工具，可以将采集的数据进行可视化的展示。如图 8-15 所示是 Grafana 官网首页，可以看出 Grafana 是做图形化分析的工具。



图 8-15　Grafana 官网首页

可以在 Grafana 官网下载页（官网下载页地址：https://grafana.com/grafana/download）下载 Grafana，里面详细介绍了各个操作系统如何安装，这里不再赘述。

8.3.3　Spring Boot 项目使用 Prometheus

新建项目，在项目中加入 micrometer-registry-prometheus 依赖，并且需要开启 actuator 端点，依赖文件如代码清单 8-32 所示。

代码清单 8-32　Prometheus 依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

在配置文件中加入配置应用名，并且开启全部 actuator 端点，如代码清单 8-33 所示。

代码清单 8-33　开启 actuator 端点代码

```
spring.application.name=chapter8-6
management.endpoints.web.exposure.include=*
```

在启动类中注入 MeterRegistryCustomizer 类，Chapter86Application 类完整内容如代码清单 8-34 所示。

代码清单 8-34　启动类代码

```
@SpringBootApplication
public class Chapter86Application {
    public static void main(String[] args) {
        SpringApplication.run(Chapter86Application.class, args);
    }

    @Value("${spring.application.name}")
    private String applicationName;

    @Bean
    MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
        return registry -> registry.config().commonTags("application",
applicationName);
    }
}
```

启动项目，访问 http://localhost:8080/actuator/prometheus，可以看到如图 8-16 所示的页面。

```
localhost:8080//actuator/prometheus

# HELP jvm_threads_peak The peak live thread count since the Java virtual machine started or peak was reset
# TYPE jvm_threads_peak gauge
jvm_threads_peak{application="chapter8-6",} 42.0
# HELP tomcat_sessions_active_current
# TYPE tomcat_sessions_active_current gauge
tomcat_sessions_active_current{application="chapter8-6",} 0.0
# HELP jvm_gc_pause_seconds Time spent in GC pause
# TYPE jvm_gc_pause_seconds summary
jvm_gc_pause_seconds_count{action="end of minor GC",application="chapter8-6",cause="Allocation Failure",} 2.0
jvm_gc_pause_seconds_sum{action="end of minor GC",application="chapter8-6",cause="Allocation Failure",} 0.05
jvm_gc_pause_seconds_count{action="end of minor GC",application="chapter8-6",cause="Metadata GC Threshold",} 1.0
jvm_gc_pause_seconds_sum{action="end of minor GC",application="chapter8-6",cause="Metadata GC Threshold",} 0.017
jvm_gc_pause_seconds_count{action="end of major GC",application="chapter8-6",cause="Metadata GC Threshold",} 1.0
jvm_gc_pause_seconds_sum{action="end of major GC",application="chapter8-6",cause="Metadata GC Threshold",} 0.115
# HELP jvm_gc_pause_seconds_max Time spent in GC pause
# TYPE jvm_gc_pause_seconds_max gauge
jvm_gc_pause_seconds_max{action="end of minor GC",application="chapter8-6",cause="Allocation Failure",} 0.0
jvm_gc_pause_seconds_max{action="end of minor GC",application="chapter8-6",cause="Metadata GC Threshold",} 0.0
jvm_gc_pause_seconds_max{action="end of major GC",application="chapter8-6",cause="Metadata GC Threshold",} 0.0
# HELP jvm_gc_memory_allocated_bytes_total Incremented for an increase in the size of the young generation memory pool after one GC to before the next
# TYPE jvm_gc_memory_allocated_bytes_total counter
jvm_gc_memory_allocated_bytes_total{application="chapter8-6",} 2.43296672E8
# HELP jvm_gc_max_data_size_bytes Max size of old generation memory pool
# TYPE jvm_gc_max_data_size_bytes gauge
jvm_gc_max_data_size_bytes{application="chapter8-6",} 1.431830528E9
# HELP jvm_gc_live_data_size_bytes Size of old generation memory pool after a full GC
# TYPE jvm_gc_live_data_size_bytes gauge
jvm_gc_live_data_size_bytes{application="chapter8-6",} 1.9048936E7
# HELP tomcat_sessions_rejected_total
# TYPE tomcat_sessions_rejected_total counter
tomcat_sessions_rejected_total{application="chapter8-6",} 0.0
# HELP jvm_buffer_memory_used_bytes An estimate of the memory that the Java virtual machine is using for this buffer pool
# TYPE jvm_buffer_memory_used_bytes gauge
jvm_buffer_memory_used_bytes{application="chapter8-6",id="direct",} 81920.0
jvm_buffer_memory_used_bytes{application="chapter8-6",id="mapped",} 0.0
# HELP tomcat_global_request_seconds
# TYPE tomcat_global_request_seconds summary
tomcat_global_request_seconds_count{application="chapter8-6",name="http-nio-8080",} 9.0
tomcat_global_request_seconds_sum{application="chapter8-6",name="http-nio-8080",} 0.617
# HELP process_files_max The maximum file descriptor count
# TYPE process_files_max gauge
process_files_max{application="chapter8-6",} 10240.0
# HELP jvm_threads_live The current number of live threads including both daemon and non-daemon threads
# TYPE jvm_threads_live gauge
jvm_threads_live{application="chapter8-6",} 28.0
# HELP tomcat_sessions_created_total
# TYPE tomcat_sessions_created_total counter
tomcat_sessions_created_total{application="chapter8-6",} 0.0
# HELP system_cpu_count The number of processors available to the Java virtual machine
# TYPE system_cpu_count gauge
system_cpu_count{application="chapter8-6",} 4.0
# HELP system_load_average_1m The sum of the number of runnable entities queued to available processors and the number of runnable entities running on the available processors averaged over a period of time
# TYPE system_load_average_1m gauge
system_load_average_1m{application="chapter8-6",} 3.259765625
# HELP tomcat_sessions_alive_max_seconds
# TYPE tomcat_sessions_alive_max_seconds gauge
```

图 8-16　查看 prometheus 端点

8.3.4　Prometheus 配置

进入 Prometheus 安装目录，打开 prometheus.yml 文件，在 scrape_configs 中加入如下配置，如代码清单 8-35 所示。

代码清单 8-35　Prometheus 配置代码

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s
alerting:
  alertmanagers:
  - static_configs:
    - targets:
rule_files:
scrape_configs:
### 监控prometheus
  - job_name: 'prometheus'
    static_configs:
    - targets: ['localhost:9090']
### 监控SpringBoot项目
  - job_name: 'chapter8-6'
    scrape_interval: 5s
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['127.0.0.1:8080']
```

启动 Prometheus，如代码清单 8-36 所示。

代码清单 8-36　启动 Prometheus 代码

```
./Prometheus
```

启动后，访问 http://localhost:9090/graph，如图 8-17 所示。

图 8-17　Prometheus 管理页面

单击导航栏 Status 中的 Targets，可以查看被 Prometheus 监控的应用，如图 8-18 所示。

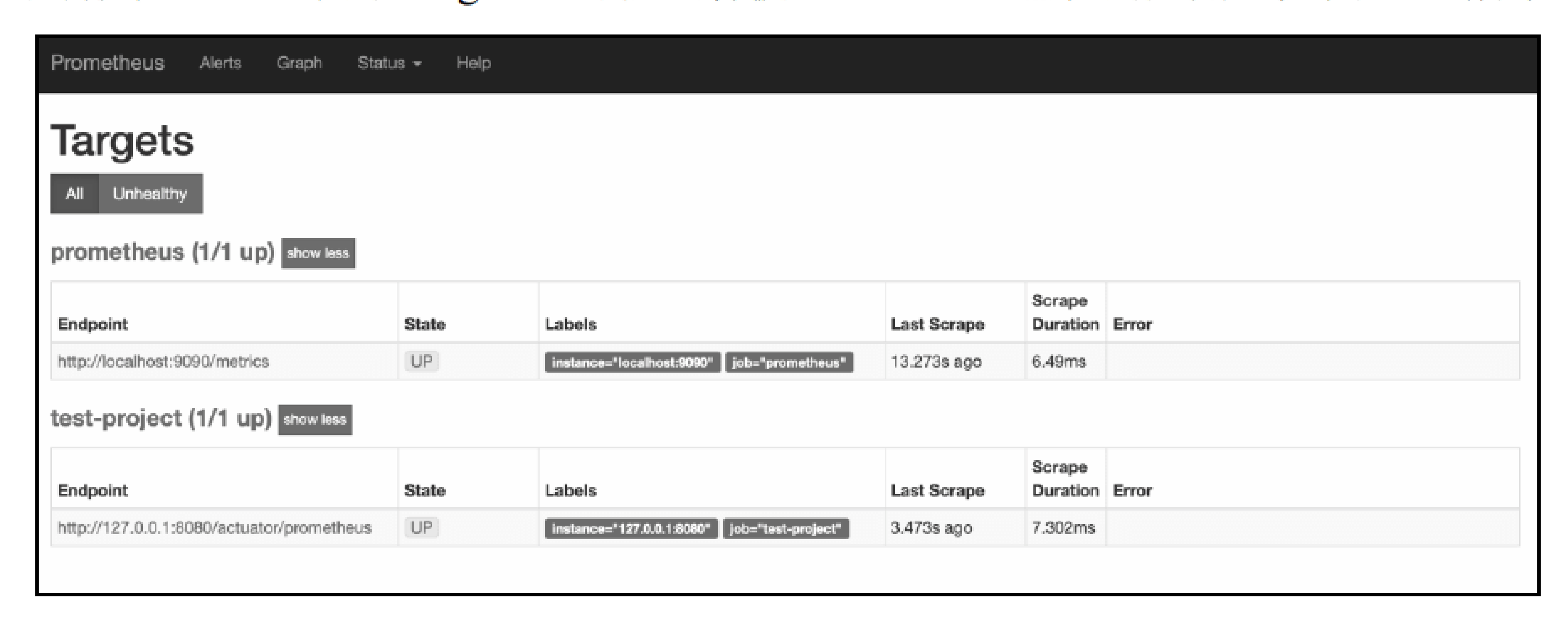


图 8-18　查看被 Prometheus 监控的页面

8.3.5　启动 Grafana

启动 Grafana，然后访问 http://localhost:3000/login，可以看到如图 8-19 所示的页面。

输入用户名 admin、密码 admin 进行登录。添加一个 Prometheus 数据源，在 URL 处配置 Prometheus 地址，因为都是本地安装，所以配置 http://localhost:9090 即可，配置完成后，单击 Save & Test 按钮，配置页如图 8-20 所示。

对于监控，这里推荐一个 Grafana 的看板，地址是 https://grafana.com/dashboards/6756，如图 8-21 所示。



图 8-19　Grafana 登录页面

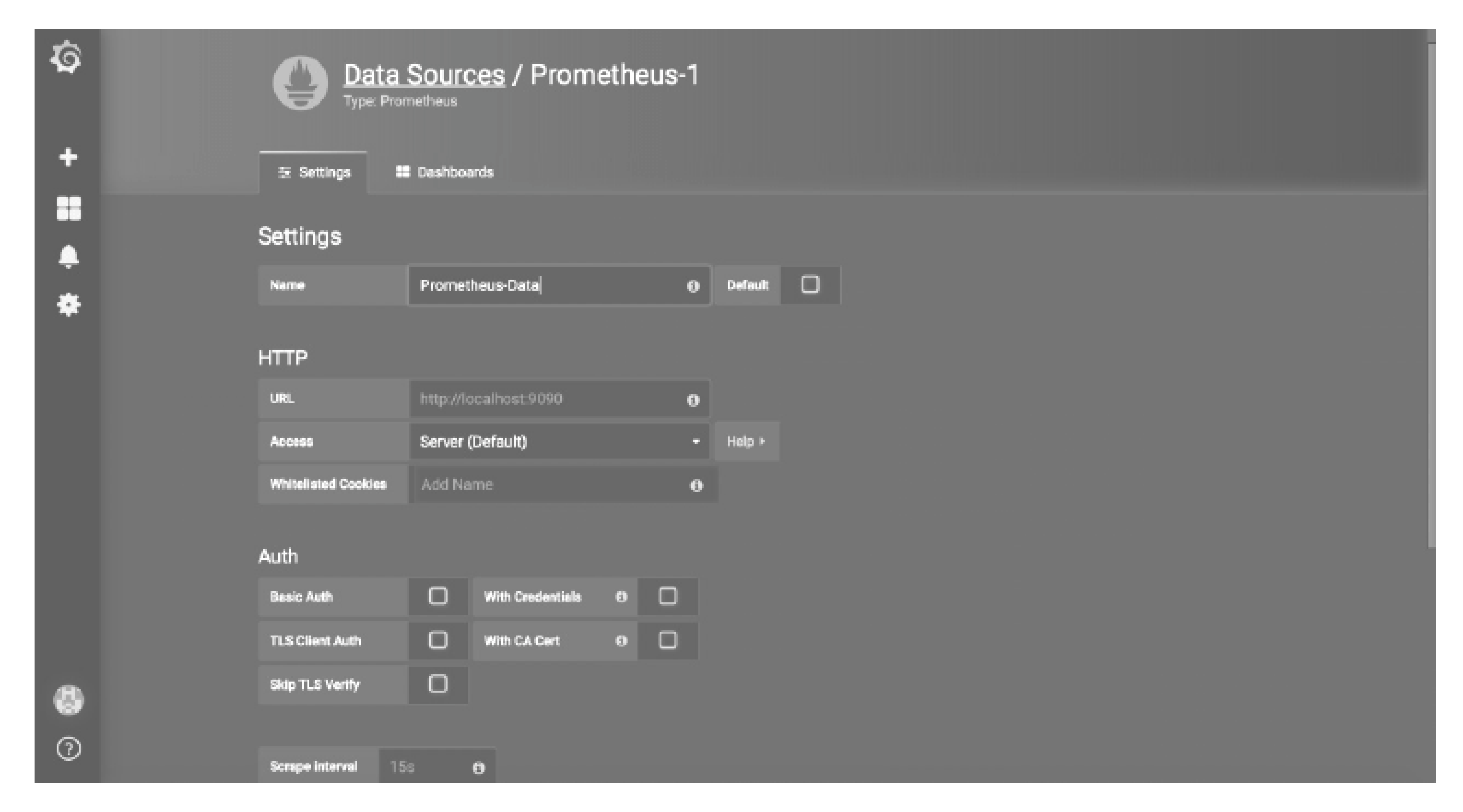


图 8-20　Grafana 配置数据源

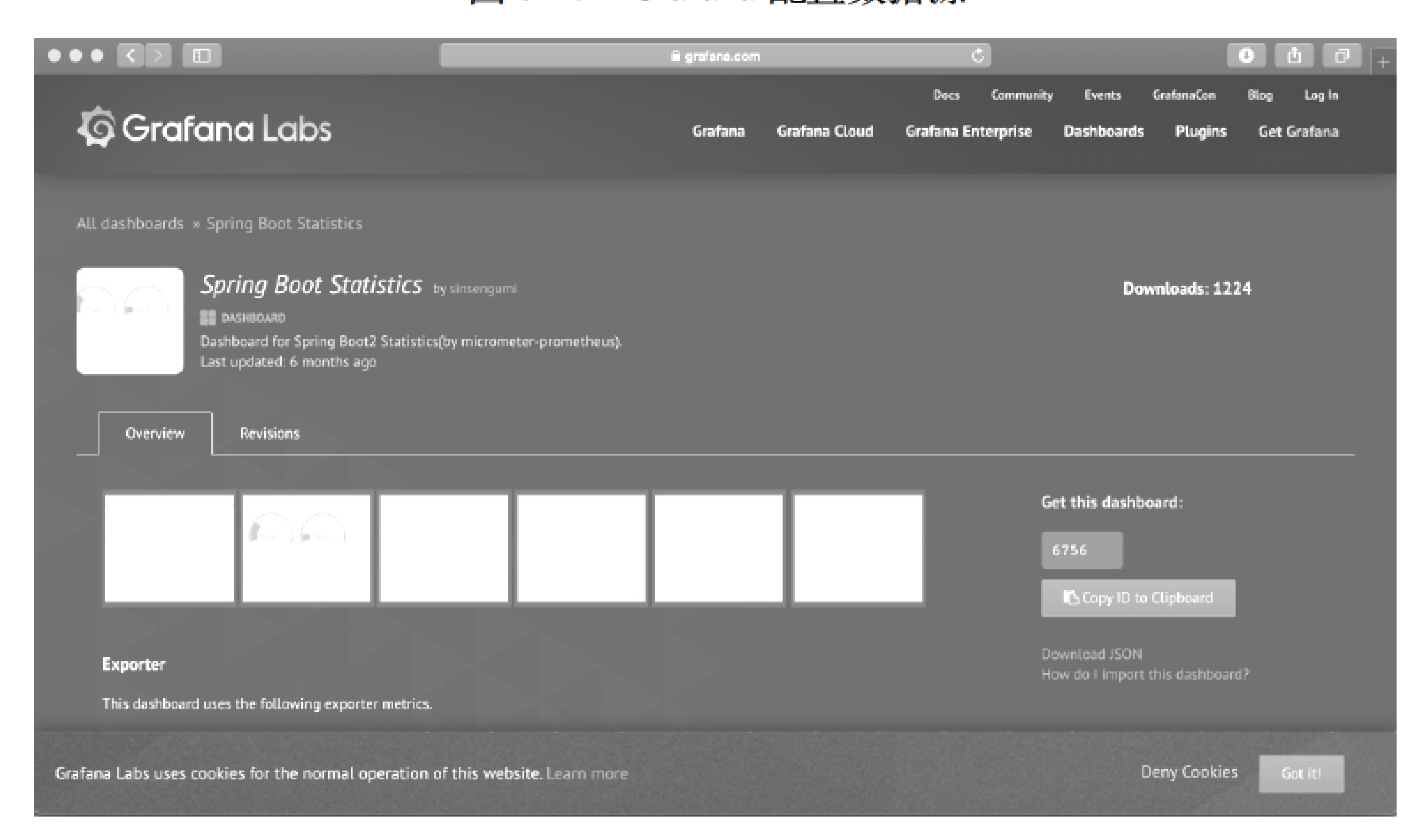


图 8-21　Grafana 查看 Dashboards 页面

单击 Grafana 导航栏的+号按钮，选择 import，这里输入 6756，然后进入如图 8-22 所示的页面。

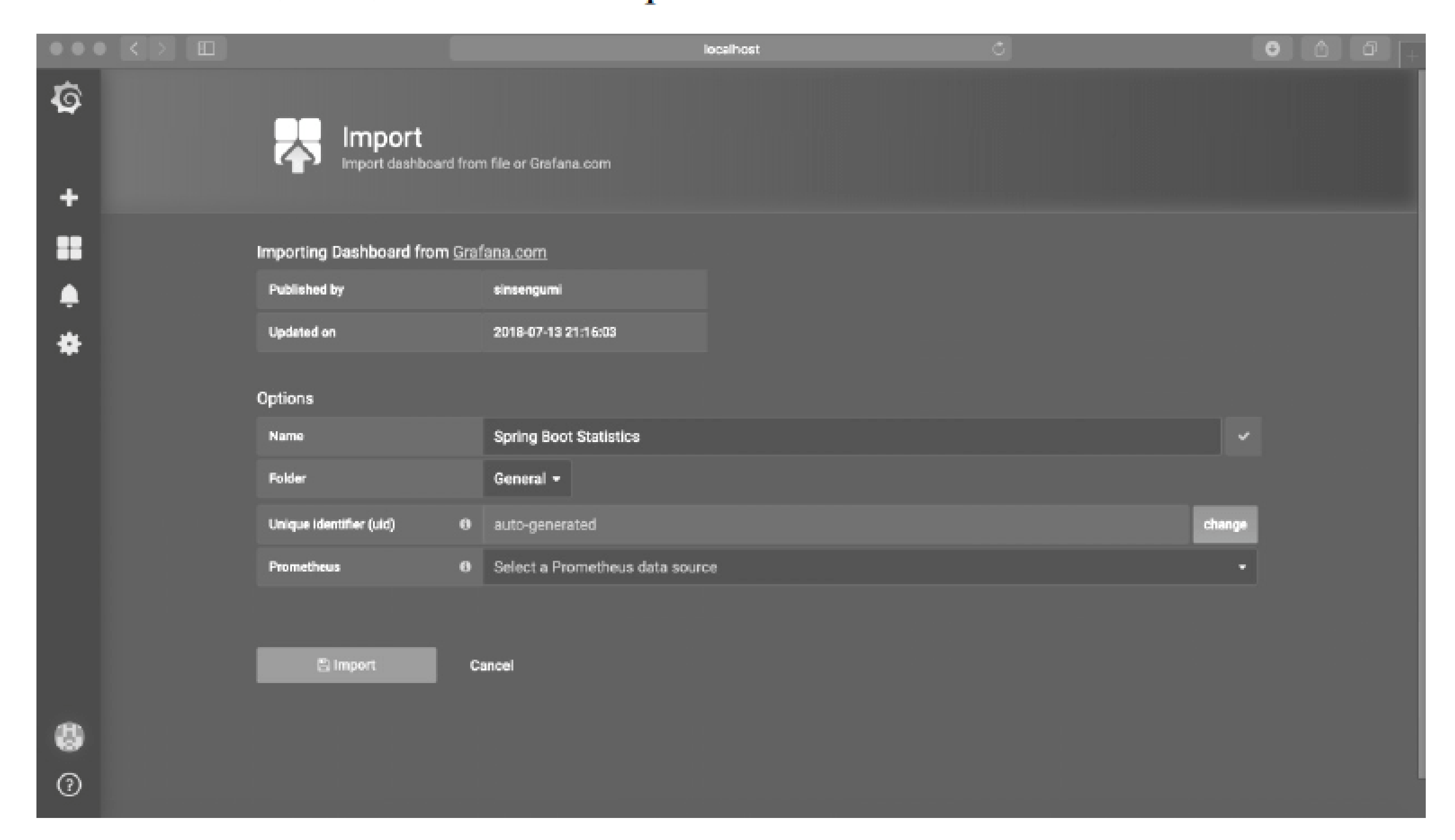


图 8-22　Grafana 导入 Dashboards 页面

Prometheus 选择刚刚添加的 Prometheus-Data，然后进入监控页面，如图 8-23 所示。

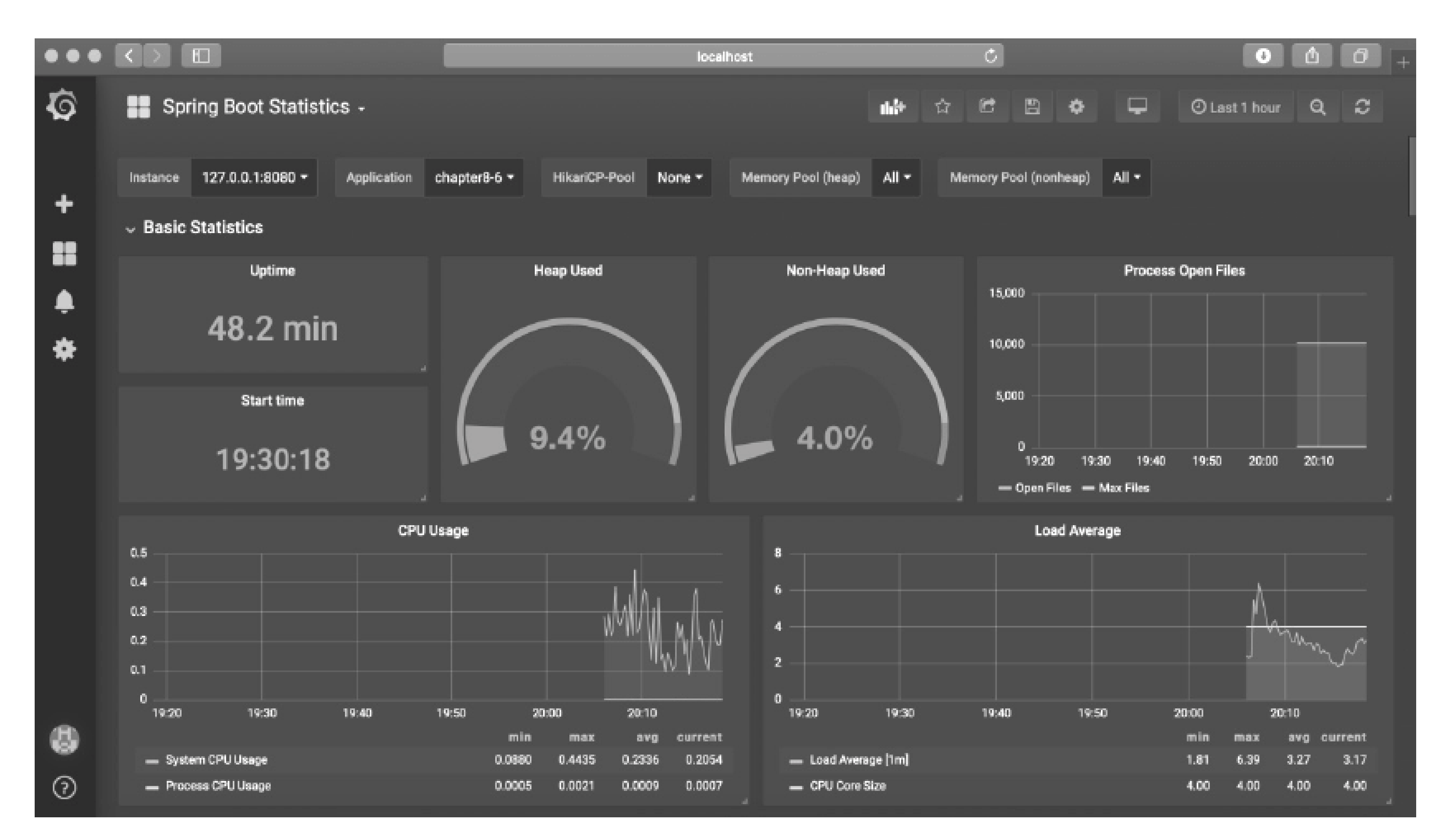


图 8-23　Grafana 查看指标页面

从图 8-23 中可以看到，已经监控到了我们想要监控的 Spring Boot 应用，这个监控页还有很多监控内容，比如 HTTP Statistics、Tomcat Statistics 等，如图 8-24 所示。

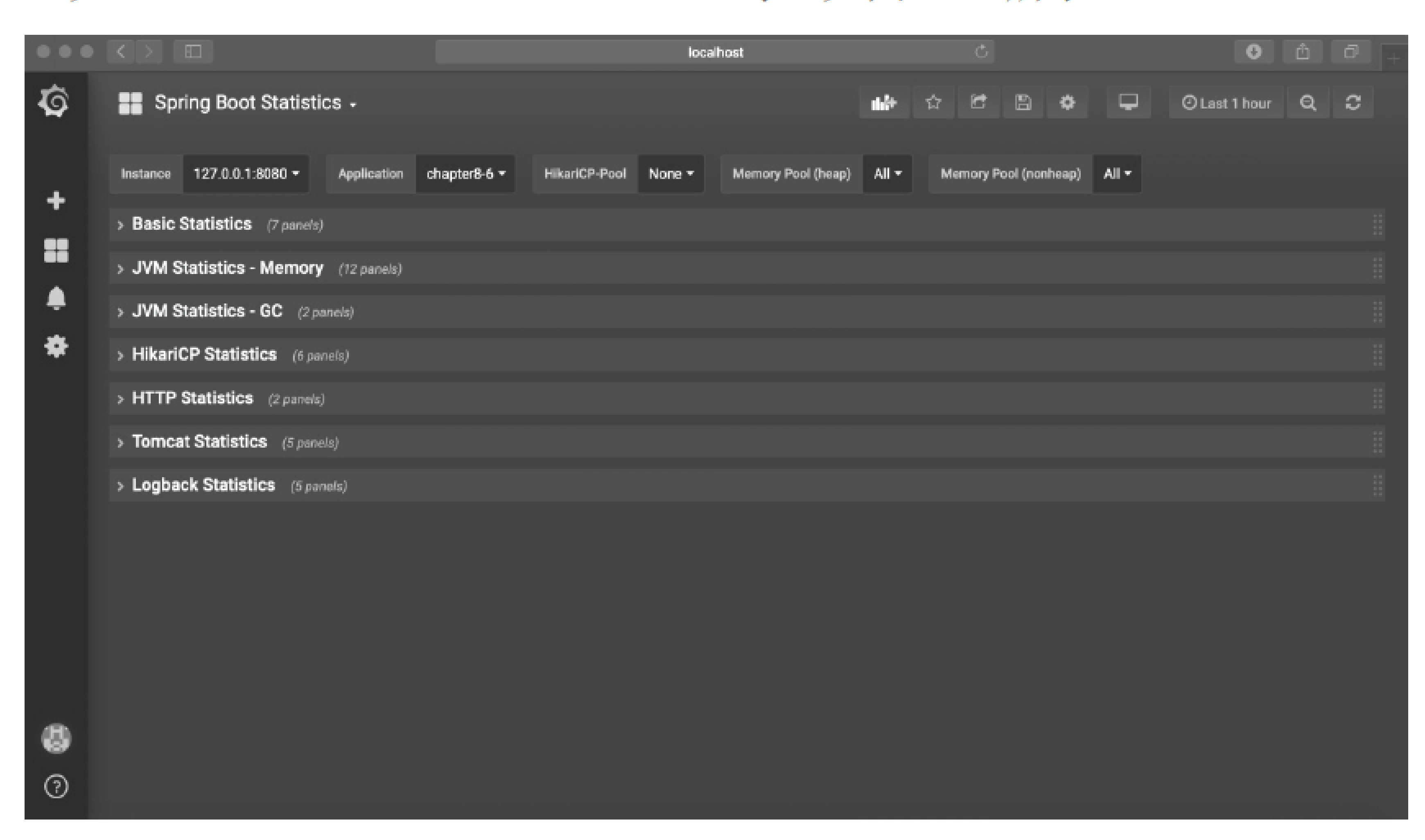


图 8-24　Grafana 监控内容

在这个监控中已经提供了足够多的监控查询，当然，在实际开发中可以添加自己需要监控的内容，根据项目场景添加即可。

8.4　小　　结

本章对 Spring Boot 服务的监控组件进行了一定的学习，从而保证服务的可用性。相信经过本章的学习，读者对监控已经有了一定了解，并且可以预防由项目意外瘫痪造成不必要的影响。

第 9 章

Spring Boot 的消息之旅

消息队列的英文名称为 Message Queue，通常简称为 MQ。MQ 是一种应用程序对应用程序的通信方法。消息队列是分布式系统中不可或缺的组件，主要解决应用解耦、异步消息、流量削峰等问题，实现高性能、高可用、可伸缩和最终一致性的架构。如今常用的开源消息队列组件有 RabbitMQ、Kafka、ActiveMQ、RocketMQ 等。

消息队列是典型的消费-生产者模式，生产者向消息队列生产消息，消费者可以从订阅的队列中读取消息。本章将对 Spring Boot 使用 RabbitMQ、Kafka、RocketMQ 消息队列进行介绍。

9.1 RabbitMQ 消息队列

RabbitMQ 是一个比较常用的消息队列，本小节将对它进行介绍，并介绍 Spring Boot 如何使用 RabbitMQ。

9.1.1 RabbitMQ 介绍

RabbitMQ 是一个由 Erlang 开发的 AMQP（Advanced Message Queuing Protocol）开源实现。很多人可能并不知道什么是 AMQP。AMQP 是一个提供统一服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息中间件设计。基于此协议的客户端与消息中间件可以传递消息，并不受客户端/中间件不同产品、不同开发语言等条件的限制。

RabbitMQ 是由 RabbitMQ Technologies Ltd 开发并且提供商业支持的。该公司在 2010 年 4 月被 SpringSource（VMWare 的一个部门）收购。在 2013 年 5 月被并入 Pivotal。其实 VMWare、Pivotal 和 EMC 本质上是一家的。不同的是，VMWare 是独立上市子公司，而 Pivotal 整合了 EMC 的某些资源，现在并没有上市。

RabbitMQ 支持多种客户端，如 Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP 等，支持 Ajax。用于分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面都有不错的表现。并且，正如 RabbitMQ 官网（官网地址：http://www.rabbitmq.com/）介绍的，RabbitMQ 在全球范围内在小型初创公司和大型企业中进行了超过 35 000 次 RabbitMQ 生产部署，是最受欢迎的开源消息代理。RabbitMQ 很轻量级，易于在内部和云中部署。它支持多种消息传递协议。RabbitMQ 可以部署在分布式和联合配置中，以满足高规模、高可用性的要求。

9.1.2　RabbitMQ 的几种角色

RabbitMQ 是一个消息代理，它的工作就是接收和转发消息。你可以把它想象成一个邮局：你把信件放入邮箱，邮递员就会把信件投递到收件人处。在这个比喻中，RabbitMQ 就扮演着邮箱、邮局以及邮递员的角色。

RabbitMQ 和邮局的主要区别在于它不处理纸张，而是接收、存储和发送消息（Message）这种二进制数据。

下面是 RabbitMQ 和消息所涉及的一些术语。

- 生产（Producing）的意思就是发送。发送消息的程序就是一个生产者（Producer）。我们一般用 P 来表示，如图 9-1 所示。

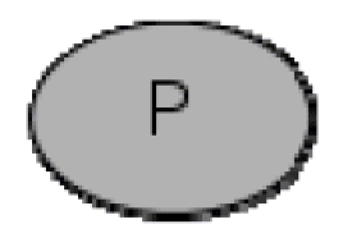


图 9-1　发送消息的生产者

- 队列（Queue）就是存在于 RabbitMQ 中邮箱的名称。虽然消息的传输经过了 RabbitMQ 和应用程序，但是它只能存储于队列中。实质上，队列就是一个巨大的消息缓冲区，它的大小只受主机内存和硬盘限制。多个生产者（Producers）可以把消息发送给同一个队列，同样，多个消费者（Consumers）也能够从同一个队列中获取数据。队列可以绘制，如图 9-2 所示。



图 9-2　队列

- 在这里，消费（Consuming）和接收（Receiving）是同一个意思。一个消费者（Consumer）就是一个等待获取消息的程序。我们把它绘制为 C，如图 9-3 所示。

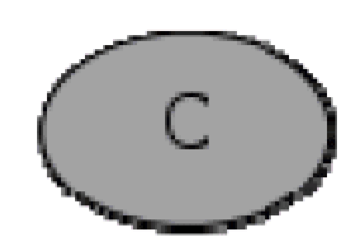


图 9-3　消费者

需要指出的是，生产者、消费者、代理不需要待在同一个设备上。事实上，大多数应用确实不会将它们放在同一台机器上。

9.1.3 RabbitMQ 的几种模式

RabbitMQ 包含几种经典的消息传递模式，下面分别进行介绍。

1. 简单模式

在使用 RabbitMQ 消息队列的时候，最容易理解的就是点对点消息发送。我们可以这样理解这种模式，比如张三给李四发送信息，首先张三编辑短信，编辑完成后发送到信息中转站，然后由中转站转发送到李四的手机里。如图 9-4 所示，P 代表生产者，C 代表消费者，中间的盒子代表消费者保留的消息缓冲区，也就是队列，这种模式多用于聊天场景。

2. 工作队列模式

在发送消息时，还有这样一种场景，就是将一个消息发送给多个消费者。如果没有消息队列，我们就只能利用 HTTP 或者其他方式对多个消费者请求数据；如果使用消息队列，我们只需要将消息推送到工作队列中就可以解决问题。

工作队列（又称任务队列，Task Queues）是为了避免等待一些占用大量资源、时间的操作。当我们把任务（Task）当作消息发送到队列中时，一个运行在后台的工作者（Worker）进程就会取出任务，然后进行处理。当你运行多个工作者（Workers）时，任务就会在它们之间共享。

这个概念在网络应用中是非常有用的，多用于资源调度或抢红包等场景，它可以在短暂的 HTTP 请求中处理一些复杂的任务。如图 9-5 所示是工作队列模式的消息流转图。

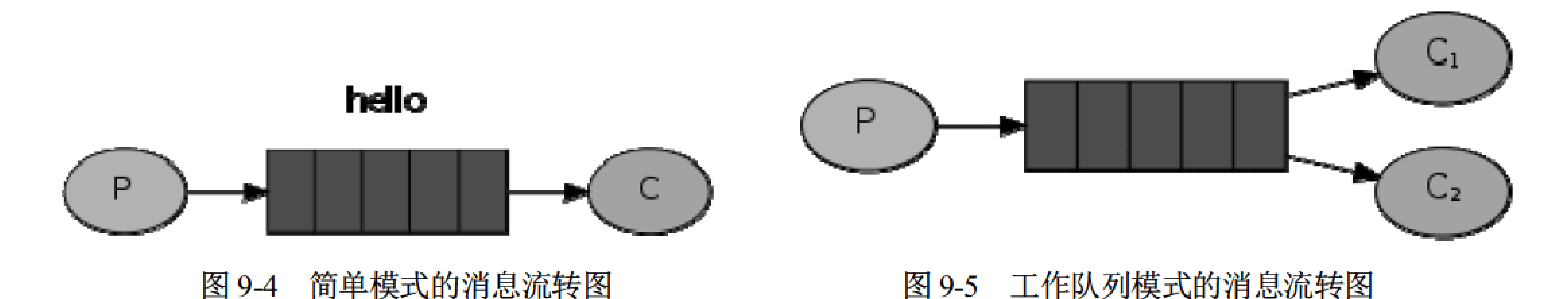


图 9-4　简单模式的消息流转图　　图 9-5　工作队列模式的消息流转图

3. 订阅模式

生产者（Producer）只需要把消息发送给一个交换机（Exchange）。交换机非常简单，它一边从生产者接收消息，一边把消息推送到队列。交换机必须知道如何处理它接收到的消息，是应该推送到指定的队列，还是推送到多个队列，或者直接忽略消息。这些规则是通过交换机类型（Exchange Type）来定义的。

有几个可供选择的交换机类型：直连（Direct）交换机、主题（Topic）交换机、头（Headers）交换机和扇型（Fanout）交换机。

可能到这里还是不好理解订阅模式与工作队列模式的区别，其实最简单的区别就是如果订阅模式有多个消费者，那么所有消费者都会收到消息，而工作队列模式只有一个消费者进行消费。订阅模式多用于广告、群聊等功能。如图 9-6 所示是订阅模式的消息流转图。

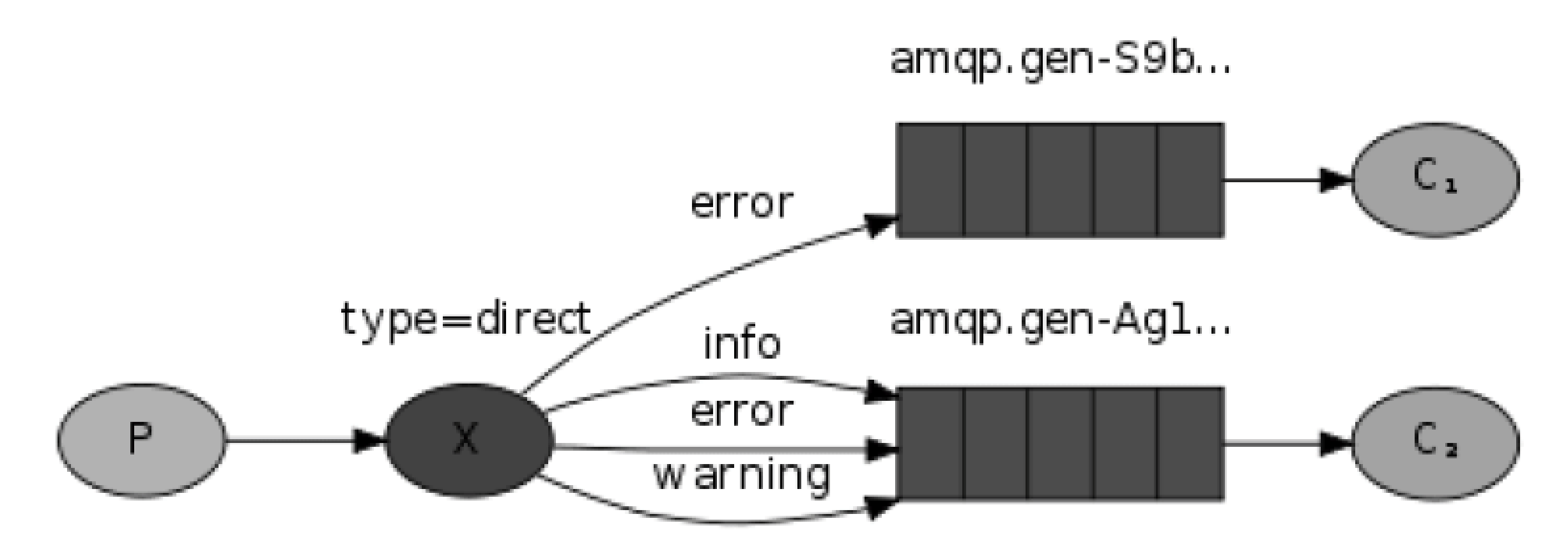


图 9-6　订阅模式的消息流转图

4. 路由模式

生产者将消息发送到交换机，在绑定队列和交换机的时候有一个路由 key，生产者发送的消息会指定路由 key，消息只会发送到 key 相同的队列，接着监听该队列消费者的消费信息。

路由模式很好理解，其实可以理解为订阅模式的特例，需要根据指定 key 来发布和订阅。一般来说，路由模式多用于项目中的报错信息。如图 9-7 所示是路由模式的消息流转图。

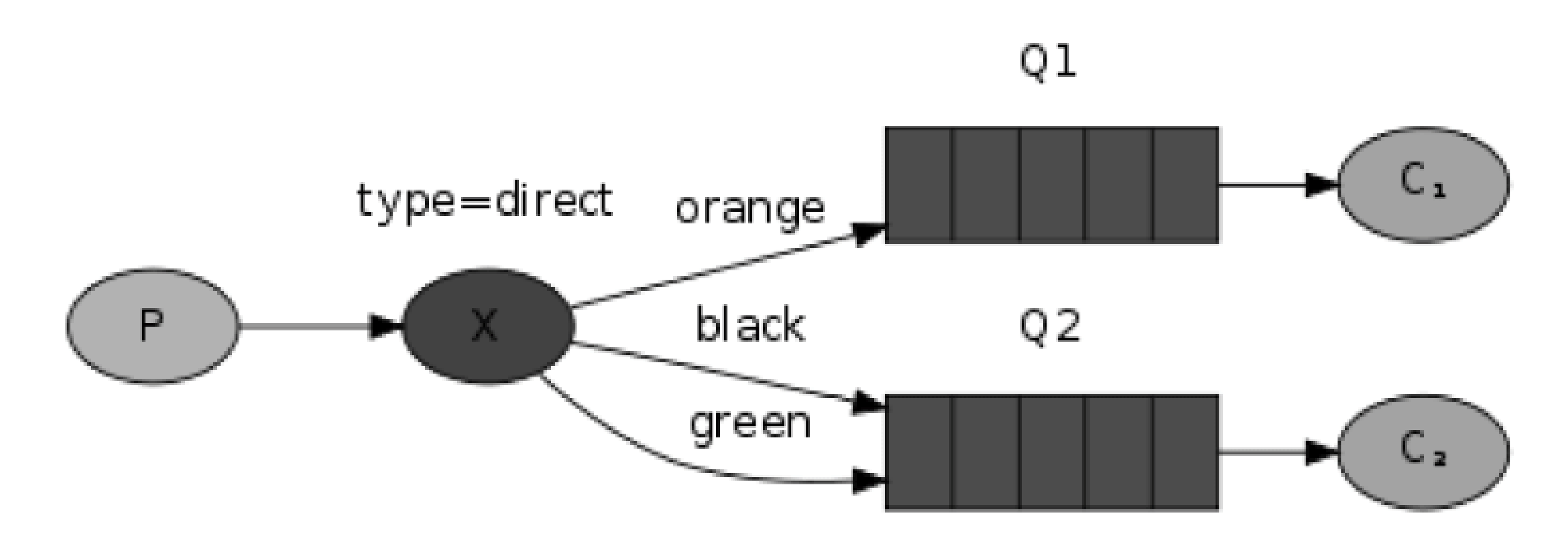


图 9-7　路由模式的消息流转图

5. topic 模式

topic 模式与路由模式大致相同，不同的是 topic 模式通过匹配符订阅多个主题的消息，比如：

- *(星号) 用来表示一个单词。
- #(井号) 用来表示任意数量（零个或多个）单词。

topic 模式的消息流转图如图 9-8 所示。

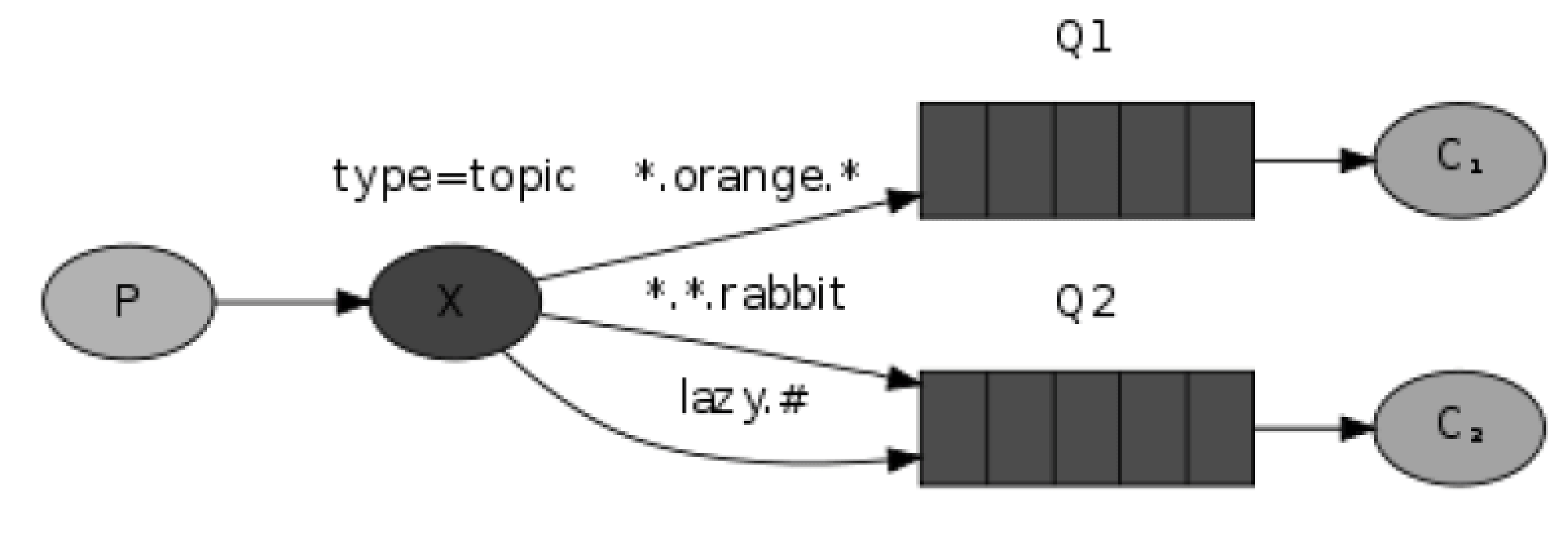


图 9-8　topic 模式的消息流转图

在这个例子里，我们发送的所有消息都是用来描述小动物的。发送的消息所携带的路由键是由 3 个单词组成的。路由键里的第一个单词描述的是动物的颜色，如图 9-8 所示的 orange（橙色）；

第二个单词是动物的种类，如图 9-8 所示的 rabbit（兔子）；第三个单词是动物的行为，如图 9-8 所示的 lazy（懒惰）。

为此，我们设置了 3 个绑定键，这 3 个绑定键可以总结为：

- Q1 队列对所有的橙色动物都感兴趣。
- Q2 队列则是对所有的兔子和所有懒惰的动物感兴趣。

举个例子，一个携带 quick.orange.rabbit 的消息将会被分别投递给 Q1 和 Q2 队列，携带 lazy.orange.elephant 的消息同样会投递给这两个队列。另一方面，携带 quick.orange.fox 的消息会投递给第一个队列，携带 lazy.brown.fox 的消息会投递给第二个队列。携带 lazy.pink.rabbit 的消息只会投递给第二个队列一次，即使它同时匹配第二个队列的两个绑定。携带 quick.brown.fox 的消息不会投递给任何一个队列。

如果我们违反约定，发送了一个携带一个单词或者 4 个单词（orange 或 quick.orange.male.rabbit）的消息，发送的消息不会投递给任何一个队列，并且会丢失掉。

但是，即使 lazy.orange.male.rabbit 有 4 个单词，还是会匹配最后一个绑定，并且投递到第二个队列。

6. 远程过程调用

RPC 是指远程过程调用。也就是说有两台服务器 A 和 B，一个应用部署在 A 服务器上，想要调用 B 服务器上应用提供的函数/方法，由于不在一个内存空间，因此不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。

一般在 RabbitMQ 中做 RPC 是很简单的。客户端发送请求消息，服务器回复响应的消息。为了接收响应的消息，我们需要在请求消息中发送一个回调队列。消息流转图如图 9-9 所示。

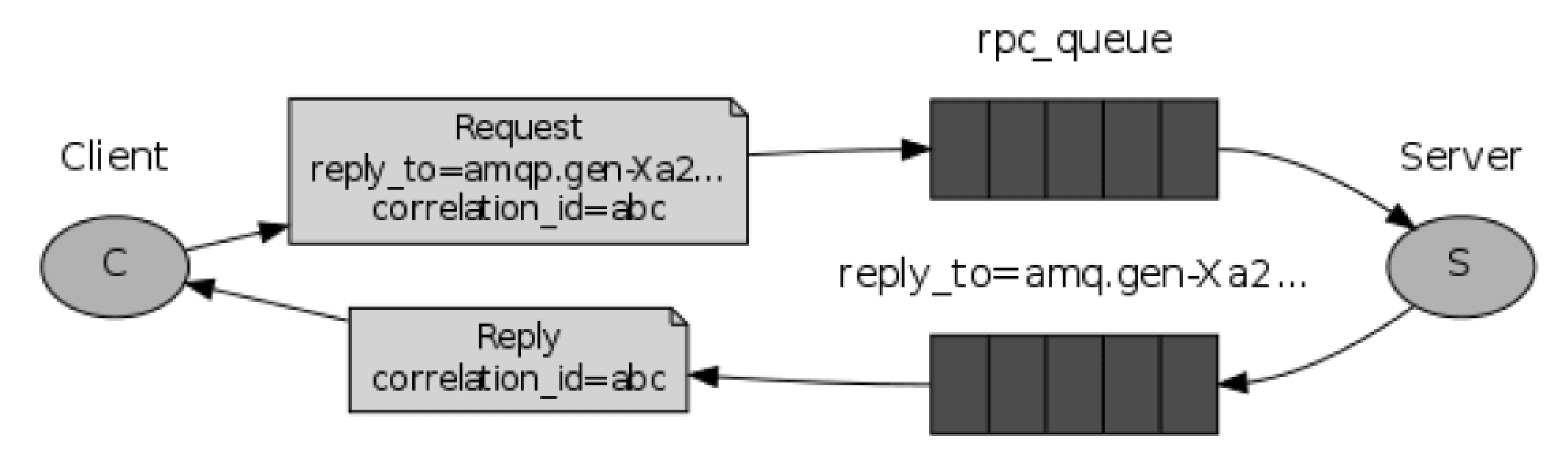


图 9-9 远程过程调用模式的消息流转图

9.1.4 Spring Boot 使用 RabbitMQ

在 Spring Boot 中使用 RabbitMQ 前需要启动 RabbitMQ。当启动 RabbitMQ 后，使用 RabbitMQ 大致分为如下几步：

（1）加入 RabbitMQ 依赖。

（2）配置 RabbitMQ 服务信息。

（3）编写消费者和生产者。

熟知上述步骤后，接下来我们新建项目，在项目中加入 spring-boot-starter-amqp 依赖，如代码清单 9-1 所示。

代码清单 9-1　RabbitMQ 项目引入 amqp 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

接下来，需要在配置文件中配置 RabbitMQ 服务信息，如代码清单 9-2 所示。

代码清单 9-2　RabbitMQ 项目配置 RabbitMQ

```
spring.rabbitmq.host=RabbitMQ 服务 IP
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=admin
```

一般来说，消息队列发送的数据都是实体对象，这里创建一个商品实体类 Goods 进行数据传输。Goods 实体内容如代码 9-3 清单所示。

代码清单 9-3　RabbitMQ 项目发送实体代码

```
public class Goods implements Serializable {
    private static final long serialVersionUID = 6629065135155452917L;
    private Long goodsId;
    private String goodsName;
    private String goodsIntroduce;
    private Double goodsPrice;

    ... //省略 set、Get 方法
}
```

这里以使用 RabbitMQ 的简单消息发送、Topic 转发模式消息发送和 Fanout Exchange 模式消息发送 3 种为例，使用 Spring Boot 操作 RabbitMQ 进行消息发送和接收。

1. 简单消息发送

简单消息发送很好理解，前面已经介绍过了，其实就是发送者将消息发送到消息队列，再由消息队列转发到消费者。首先创建一个简单发送消息配置 DirectConfig 类，在类上加入注解 @Configuration，表明这是一个配置类。在类中定义一个点对点消息发送的常量值用作消息队列名称，同时向 Spring 注入 Queue 类并创建消息队列，完整内容如代码清单 9-4 所示。

代码清单 9-4 RabbitMQ 项目简单消息发送配置

```
@Configuration
public class DirectConfig {
    public static final String DIRECT_QUEUE = "direct.queue";

    @Bean
    public Queue directQueue() {
        // 第一个参数是队列名字， 第二个参数是指是否持久化
        return new Queue("direct.queue", true);
    }
}
```

接下来创建一个消息发送者 DirectSender 类。一般使用消息发送都是通过操作 AmqpTemplate 类，所以首先注入这个类。然后创建一个发送消息的方法，调用 convertAndSend()方法进行消息发送。DirectSender 类完整内容如代码清单 9-5 所示。

代码清单 9-5 RabbitMQ 项目简单消息发送生产者

```
@Component
public class DirectSender {
    private static final Logger log =
LoggerFactory.getLogger(DirectSender.class);

    @Autowired
    private AmqpTemplate amqpTemplate;

    public void sendDirectQueue() {
        Goods goods = new Goods(System.currentTimeMillis(),"测试商品","这是一
个测试的商品",98.6);
        log.info("简单消息已经发送");
        // 第一个参数是指要发送到哪个队列，第二个参数是指要发送的内容
        this.amqpTemplate.convertAndSend(DirectConfig.DIRECT_QUEUE, goods);
    }
}
```

消息发送者已经创建好了。接下来创建一个消息接收者 DirectReceiver 类，使用@RabbitListener 设置监听队列的名称，方法的参数就是接收到的实体对象，完整内容如代码清单 9-6 所示。

代码清单 9-6 RabbitMQ 项目简单消息发送消费者

```
@Component
public class DirectReceiver {
    private static final Logger log = LoggerFactory.getLogger
(DirectReceiver.class);

    // queues 是指要监听的队列的名字
```

```
    @RabbitListener(queues = DirectConfig.DIRECT_QUEUE)
    public void receiverDirectQueue(Goods goods) {
        log.info("简单消息接受成功，参数是：" + goods.toString());
    }
}
```

最后创建一个 DirectController 类进行测试，这里仅调用 directSender 生产者发送消息的方法，完整内容如代码清单 9-7 所示。

代码清单 9-7　RabbitMQ 项目测试类

```
@RestController
public class DirectController {
    @Autowired
    private DirectSender directSender;

    @GetMapping("directTest")
    public void directTest() {
        directSender.sendDirectQueue();
    }
}
```

启动应用，在浏览器中访问 http://localhost:8080/directTest，可以看到控制台如图 9-10 所示。

```
2019-01-20 11:25:46.769  INFO 6554 --- [nio-8080-exec-1] com.springboot.sender.DirectSender       : 简单消息已经发送
2019-01-20 11:25:46.874  INFO 6554 --- [cTaskExecutor-1] com.springboot.receiver.DirectReceiver   : 简单消息接受成功, 参数是: Goods{goodsId=1547954746769, goodsName='测
 试商品', goodsIntroduce='这是一个测试的商品', goodsPrice=98.6}
```

图 9-10　简单消息发送控制台

我们再来查看一下 RabbitMQ 管理页面，如图 9-11 所示。

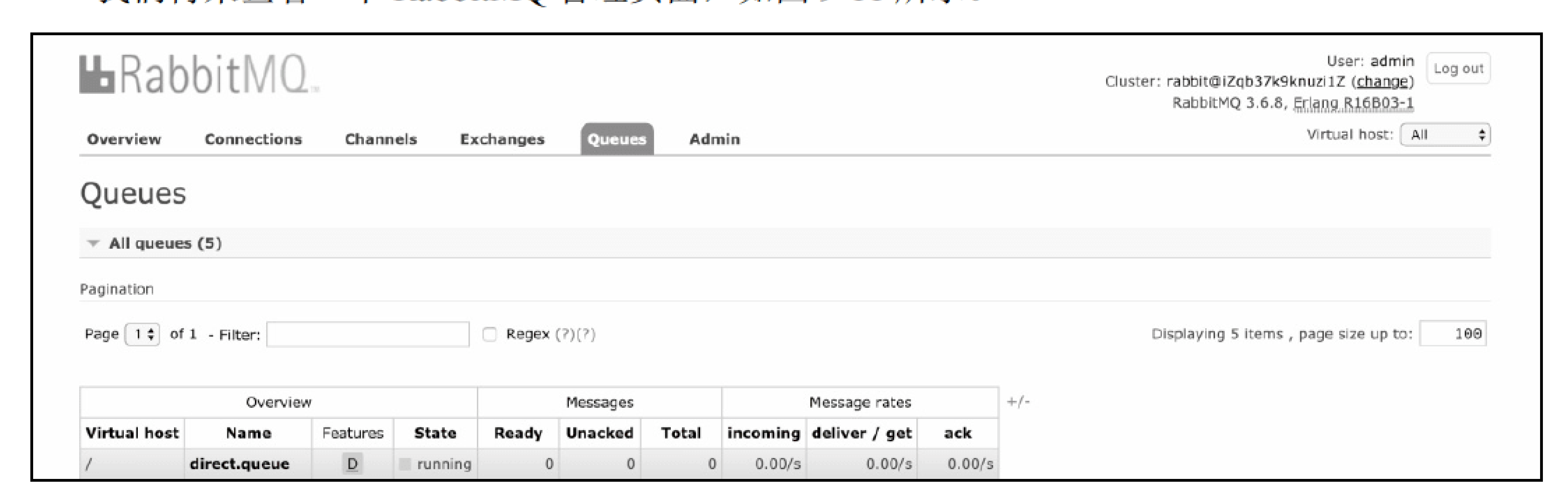


图 9-11　简单消息发送 RabbitMQ 管理页面

刚刚创建的队列在 RabbitMQ 管理页面也可以查看。至此，简单消息发送成功。

2. Topic 转发模式消息发送

Topic 转发模式是通过设置主题的方式来进行消息发送和接收的，这里需要使用到 Route-key，创建一个 TopicConfig 类配置主题和交换机，完整内容如代码清单 9-8 所示。

代码清单 9-8 RabbitMQ 项目 Topic 模式配置

```
@Configuration
public class TopicConfig {
    public static final String TOPIC_QUEUE1 = "topic.queue1";
    public static final String TOPIC_QUEUE2 = "topic.queue2";
    public static final String TOPIC_EXCHANGE = "topic.exchange";

    @Bean
    public Queue topicQueue1() {
        return new Queue(TOPIC_QUEUE1);
    }
    @Bean
    public Queue topicQueue2() {
        return new Queue(TOPIC_QUEUE2);
    }
    @Bean
    public TopicExchange topicExchange() {
        return new TopicExchange(TOPIC_EXCHANGE);
    }
    @Bean
    public Binding topicBinding1() {
        return BindingBuilder.bind(topicQueue1()).to(topicExchange()).
with("topic.messge");
    }
    @Bean
    public Binding topicBinding2() {
        return BindingBuilder.bind(topicQueue2()).to(topicExchange()).
with("topic.#");
    }
}
```

生产者发送消息还是使用 convertAndSend()方法，不过需要在参数内设置 Route-key，完整内容如代码清单 9-9 所示。

代码清单 9-9 RabbitMQ 项目 Topic 模式生产者

```
@Component
public class TopicSender {
    private static final Logger log =
LoggerFactory.getLogger(TopicSender.class);
    @Autowired
    private AmqpTemplate amqpTemplate;

    public void sendTopicQueue() {
```

```
        Goods goods1 = new Goods(System.currentTimeMillis(),"测试商品 1","这是
第一个测试的商品",98.6);
        Goods goods2 = new Goods(System.currentTimeMillis(),"测试商品 2","这是
第二个测试的商品",100.0);
        log.info("TopicSender 已发送消息");
        // 第一个参数：TopicExchange 名字
        // 第二个参数：Route-Key
        // 第三个参数：要发送的内容
        this.amqpTemplate.convertAndSend(TopicConfig.TOPIC_EXCHANGE,
"topic.messge", goods1 );
        this.amqpTemplate.convertAndSend(TopicConfig.TOPIC_EXCHANGE,
"topic.messge2", goods2);
      }
   }
```

这里其实是创建两个消费者，分别订阅不同主题的内容。测试的目的很简单，消费者 2 订阅的主题包含生产者发送的两条信息，消费者 1 只能收到其中的一条信息。消费者完整内容如代码清单 9-10 所示。

代码清单 9-10　RabbitMQ 项目 topic 模式消费者

```
   @Component
   public class TopicReceiver {
      private static final Logger log = LoggerFactory.getLogger
(TopicReceiver.class);

      @RabbitListener(queues = TopicConfig.TOPIC_QUEUE1)
      public void receiveTopic1(Goods goods) {
         log.info("receiveTopic1 收到消息：" + goods.toString());
      }
      @RabbitListener(queues = TopicConfig.TOPIC_QUEUE2)
      public void receiveTopic2(Goods goods) {
         log.info("receiveTopic2 收到消息：" + goods.toString());
      }
   }
```

创建 TopicController 类进行调用测试，如代码清单 9-11 所示。

代码清单 9-11　RabbitMQ 项目 Topic 模式测试类

```
   @RestController
   public class TopicController {
      @Autowired
      private TopicSender topicSender;

      @GetMapping("topicTest")
```

```
    public void topicTest() {
        topicSender.sendTopicQueue();
    }
}
```

在浏览器中访问 http://localhost:8080/topicTest，查看控制台，如图 9-12 所示。至此，Topic 转发模式消息发送已经完成。

```
2019-01-20 11:46:20.691  INFO 6663 --- [nio-8080-exec-5] com.springboot.sender.TopicSender        : TopicSender已发送消息
2019-01-20 11:46:20.813  INFO 6663 --- [cTaskExecutor-1] com.springboot.receiver.TopicReceiver    : receiveTopic2收到消息: Goods{goodsId=1547955980691, goodsName='测
 试商品1', goodsIntroduce='这是第一个测试的商品', goodsPrice=98.6}
2019-01-20 11:46:20.815  INFO 6663 --- [cTaskExecutor-1] com.springboot.receiver.TopicReceiver    : receiveTopic2收到消息: Goods{goodsId=1547955980691, goodsName='测
 试商品2', goodsIntroduce='这是第二个测试的商品', goodsPrice=100.0}
2019-01-20 11:46:20.813  INFO 6663 --- [cTaskExecutor-1] com.springboot.receiver.TopicReceiver    : receiveTopic1收到消息: Goods{goodsId=1547955980691, goodsName='测
 试商品1', goodsIntroduce='这是第一个测试的商品', goodsPrice=98.6}
```

图 9-12　Topic 模式控制台

3. Fanout Exchange 模式消息发送

Fanout Exchange 模式消息发送是指广播型消息发送，订阅了这个交换机的消息都会收到消息内容。Fanout Exchange 模式与 Topic 模式有些类似，但是不需要设置 Route-key。完整内容如代码清单 9-12 所示。

代码清单 9-12　RabbitMQ 项目 Fanout Exchange 模式配置

```
@Configuration
public class FanoutConfig {
    public static final String FANOUT_QUEUE1 = "fanout.queue1";
    public static final String FANOUT_QUEUE2 = "fanout.queue2";
    public static final String FANOUT_EXCHANGE = "fanout.exchange";
    @Bean
    public Queue fanoutQueue1() {
        return new Queue(FANOUT_QUEUE1);
    }
    @Bean
    public Queue fanoutQueue2() {
        return new Queue(FANOUT_QUEUE2);
    }
    @Bean
    public FanoutExchange fanoutExchange() {
        return new FanoutExchange(FANOUT_EXCHANGE);
    }
    @Bean
    public Binding fanoutBinding1() {
        return BindingBuilder.bind(fanoutQueue1()).to(fanoutExchange());
    }
    @Bean
    public Binding fanoutBinding2() {
```

```
        return BindingBuilder.bind(fanoutQueue2()).to(fanoutExchange());
    }
}
```

消息发送者依然调用 convertAndSend()方法。这里需要注意，第二个参数设置为空。因为 Fanout 交换机不处理路由键，只是简单地将队列绑定到交换机上，每个发送到交换机的消息都会被转发到与该交换机绑定的所有队列上。生产者内容如代码清单 9-13 所示。

代码清单 9-13 RabbitMQ 项目 Fanout Exchange 模式生产者

```
@Component
public class FanoutSender {
    private static final Logger log = LoggerFactory.getLogger
(FanoutSender.class);
    @Autowired
    private AmqpTemplate amqpTemplate;

    public void sendFanoutQueue() {
        Goods goods = new Goods(System.currentTimeMillis(),"测试商品","这是一
个测试的商品",98.6);
        log.info("sendFanoutQueue 已发送消息");
        this.amqpTemplate.convertAndSend(FanoutConfig.FANOUT_EXCHANGE, "",
goods );
    }
}
```

消费者只是创建两个消息队列来接收消息，因为使用的是同一个交换机，所以都会收到消息。消费者内容如代码清单 9-14 所示。

代码清单 9-14 RabbitMQ 项目 Fanout Exchange 模式消费者

```
@Component
public class FanoutReceiver {
    private static final Logger log =
LoggerFactory.getLogger(FanoutReceiver.class);

    @RabbitListener(queues = FanoutConfig.FANOUT_QUEUE1)
    public void receiveFanout1(Goods goods) {
        log.info("receiveFanoutQueue1 监听到消息：" + goods.toString());
    }
    @RabbitListener(queues = FanoutConfig.FANOUT_QUEUE2)
    public void receiveFanout2(Goods goods) {
        log.info("receiveFanoutQueue2 监听到消息：" + goods.toString());
    }
}
```

创建 FanoutController 来调用消息发送，如代码清单 9-15 所示。

代码清单 9-15 RabbitMQ 项目 fanout 模式测试类

```
@RestController
public class FanoutController {
    @Autowired
    private FanoutSender fanoutSender;

    @GetMapping("fanoutTest")
    public void fanoutTest(){
        fanoutSender.sendFanoutQueue();
    }
}
```

在浏览器中访问 http://localhost:8080/fanoutTest，查看控制台，如图 9-13 所示。到这里，Fanout Exchange 模式消息发送已经完成。

```
2019-01-20 12:04:43.197  INFO 6723 --- [nio-8080-exec-4] com.springboot.sender.FanoutSender       : sendFanoutQueue已发送消息
2019-01-20 12:04:43.237  INFO 6723 --- [cTaskExecutor-1] com.springboot.receiver.FanoutReceiver   : receiveFanoutQueue2监听到消息: Goods{goodsId=1547957083197,
 goodsName='测试商品', goodsIntroduce='这是一个测试的商品', goodsPrice=98.6}
2019-01-20 12:04:43.237  INFO 6723 --- [cTaskExecutor-1] com.springboot.receiver.FanoutReceiver   : receiveFanoutQueue1监听到消息: Goods{goodsId=1547957083197,
 goodsName='测试商品', goodsIntroduce='这是一个测试的商品', goodsPrice=98.6}
```

图 9-13 Fanout Exchange 模式控制台

RabbitMQ 相关内容到这里就介绍完了。由于篇幅原因，不能面面俱到，只是列举了 3 个常用的方式来供读者参阅。

9.2 Kafka 消息队列

Kafka 是由 Apache 软件基金会开发的一个开源流处理平台，被誉为最高吞吐量的常用消息队列。本节我们来学习 Spring Boot 如何使用 Kafka 消息队列。

9.2.1 Kafka 介绍

Kafka（官网地址：http://kafka.apache.org/）是一种高吞吐量的分布式发布订阅消息系统，最初由 Linkedin 公司开发，于 2010 年底在 Github 首次开源，初始版本为 0.7.0。在 2011 年 7 月，Linkedin 公司将 Kafka 项目贡献给 Apache，成为 Apache 的孵化项目，在 2012 年 10 月，Kafka 从 Apache 孵化器正式毕业，成为 Apache 顶级项目。在 2014 年，为了 Kafka 更好地发展，几名 Kafka 的核心开发人员离开了 Linkedin，成交了 Confluent 公司，继续推进 Kafka 的发展。

在业界，Kafka 无疑是最高吞吐量的分布式流处理平台，并且它也是一个优秀的分布式 MQ，其通过 Zookeeper 实现了高可用。同时，Kafka 可以做的事情非常多，下面详细介绍。

1. 消息

Kafka 可以很好地替代传统的消息代理。消息代理的使用有多种场景，如将数据处理与数据生成器分离、缓冲未处理的消息等。与大多数消息传递系统相比，Kafka 具有更好的吞吐量、内置分区、复制和容错功能，这使其成为大规模消息处理应用程序的理想解决方案。通常消息传递使用较低的吞吐量，但可能要求较低的端到端延迟，Kafka 提供强大的持久性来满足这一要求。

在这个领域，Kafka 可与传统的消息传递系统（如 ActiveMQ 或 RabbitMQ）相媲美。

2. 网站活动跟踪

Kafka 的初始用例是能够将用户活动跟踪管道重建为一组实时发布 - 订阅源。这意味着网站活动（页面查看、搜索或用户可能采取的其他操作）将发布到中心主题，每个活动类型包含一个主题。这些订阅源提供了一系列用例，包括实时处理、实时监控以及加载到 Hadoop 或离线数据仓库系统以进行离线处理和报告。因为每个用户页面视图生成了许多活动消息，活动跟踪的数据量通常非常大。

3. 度量

Kafka 通常用于监控数据。这涉及从分布式应用程序聚合统计信息，并且从中生成可操作的集中数据源。

4. 日志聚合

许多人使用 Kafka 作为日志聚合解决方案的替代品。日志聚合通常从服务器收集物理日志文件，并将它们放在中央位置（可能是文件服务器或 HDFS）进行处理。Kafka 从这些日志中提取信息，并将日志或事件数据更清晰地抽象为消息流。这样可以更低延迟的处理并更容易支持多个数据源和分布式数据消耗。与 Scribe 或 Flume 等以日志为中心的系统相比，Kafka 提供了同样出色的性能，由于复制而具有更强的耐用性保证，以及更低的端到端延迟。

5. 流处理

许多 Kafka 用户在处理由多个阶段组成的管道时处理数据，其中原始输入数据从 Kafka 主题中消费，然后聚合、修饰或通过其他方式转换为新主题，以供进一步消费或后续处理。例如，用于推荐新闻文章的处理管道可以从 RSS 订阅源抓取文章内容并将其发布到“文章”主题；进一步处理可能会对此内容进行规范化或把重复数据删除，并将已清理的文章内容发布到新主题；最终处理阶段可能会尝试向用户推荐此内容。此类处理管道基于各个主题创建实时数据流的图形。从 0.10.0.0 版本开始，这是一个轻量级但功能强大的流处理库，名为 Kafka Streams，在 Apache Kafka 中可用于执行上述数据处理。除了 Kafka Streams 之外，其他开源流处理工具包括 Apache Storm 和 Apache Samza。

6. 活动采购

事件源是一种应用程序设计风格，按照时间来记录状态的更改。Kafka 可以存储非常多的日志数据，使其成为以这种风格构建的应用程序的出色后端。

7. 提交日志

Kafka 可以从外部为分布式系统提供日志提交功能。该日志有助于在节点之间复制数据，采用重新同步机制可以从失败的节点恢复数据。Kafka 中的日志压缩功能有助于支持此用法。在这种用法中，Kafka 类似于 Apache BookKeeper 项目。

9.2.2 Spring Boot 使用 Kafka

在 Spring Boot 中使用 Kafka 的过程与使用 RabbitMQ 大致一致，分为如下几步：

（1）加入 Kafka 依赖。
（2）配置 Kafka 服务信息。
（3）编写消费者和生产者。

在使用前需要安装 Kafka 服务。本节 Spring Boot 使用 Kafka 消息队列仅以发送实体对象为例，发送的实体还是 9.1 节使用的商品实体 Goods 类。新建项目，首先在 pom 文件中加入 Kafka 依赖，如代码清单 9-16 所示。

代码清单 9-16　Kafka 项目依赖代码

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

接下来，在配置文件中配置 Kafka 生产者和消费者的信息，如代码清单 9-17 所示。

代码清单 9-17　Kafka 项目配置文件

```
### producer 配置
spring.kafka.producer.bootstrap-servers=Kafka 服务地址:9092

### consumer 配置
spring.kafka.consumer.bootstrap-servers=Kafka 服务地址:9092
spring.kafka.consumer.group-id=goods
spring.kafka.consumer.enable-auto-commit=true
spring.kafka.consumer.auto-offset-reset=latest

spring.kafka.template.default-topic=goods
```

创建一个生产者 KafkaSender 类，在 Kafka 消息队列中使用消息发送的时候操作的都是 KafkaTemplate 类。首先注入这个类，然后调用 send 方法，定义主题为 goods，完整内容如代码清单 9-18 所示。

代码清单 9-18　Kafka 项目生产者代码

```
@Component
public class KafkaSender {
    private static final Logger log = LoggerFactory.getLogger
(KafkaSender.class);
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void send() {
        Goods goods = new Goods(System.currentTimeMillis(),"测试商品","这是一
个测试的商品",98.6);
        log.info("KafkaSender 已发送消息");
        kafkaTemplate.send("goods", goods.toString());
    }
}
```

创建一个消费者 KafkaReceiver 类，使用 Kafka 消息队列消费者的时候，其实是使用 @KafkaListener 注解来监听对应的主题。经过 9.1 节的学习，读者应该大致有了思路，这里只不过是使用的类或注解不同而已。完整 KafkaReceiver 类内容如代码清单 9-19 所示。

代码清单 9-19　Kafka 项目消费者

```
@Component
public class KafkaReceiver {
    private static final Logger log = LoggerFactory.getLogger
(KafkaReceiver.class);

    @KafkaListener(topics = "goods")
    public void send(ConsumerRecord<?, ?> record) {
        Optional<?> kafkaMessage = Optional.ofNullable(record.value());
        if (kafkaMessage.isPresent()) {
            Object messge = kafkaMessage.get();
            log.info("【KafkaListener 监听到消息】" + messge);
        }
    }
}
```

最后创建一个 KafkaController 类进行调用消息发送，如代码清单 9-20 所示。

代码清单 9-20　Kafka 项目测试类代码

```
@RestController
public class KafkaController {
    @Autowired
    private KafkaSender kafkaSender;
```

```
        @GetMapping("testKafka")
        public void testKafka(){
            kafkaSender.send();
        }
    }
```

启动项目，在浏览器中访问 http://localhost:8080/testKafka，然后查看控制台，如图 9-14 所示。

```
2019-01-20 12:50:21.989  INFO 6877 --- [nio-8080-exec-1] o.a.kafka.common.utils.AppInfoParser      : Kafka version : 1.0.1
2019-01-20 12:50:21.989  INFO 6877 --- [nio-8080-exec-1] o.a.kafka.common.utils.AppInfoParser      : Kafka commitId : c0518aa65f25317e
2019-01-20 12:50:22.590  INFO 6877 --- [ntainer#0-0-C-1] com.springboot.receiver.KafkaReceiver     : 【KafkaListener监听到消息】Goods{goodsId=1547959821955,
 goodsName='测试商品', goodsIntroduce='这是一个测试的商品', goodsPrice=98.6}
```

图 9-14　Kafka 项目控制台输出

到这里，使用 Kafka 消息队列已经完成了。其实 Kafka 可以做的不只是消息，还有很多方面的使用，具体内容可以查阅官网来进行学习。

9.3　RocketMQ 消息队列

9.3.1　RocketMQ 介绍

Apache RocketMQ（官网地址：http://rocketmq.apache.org）是由阿里巴巴集团开源的大型消息队列，现在已经贡献给了 Apache 开源基金会，同时是一个分布式消息传递和流媒体平台，具有低延迟、高性能、可靠性、万亿级容量和灵活的可扩展性。

接下来介绍 RocketMQ（Github 官网地址：https://github.com/apache/rocketmq）的 Github，它提供了多种功能：

- 发布/订阅消息模型。
- 定时的消息传递。
- 按时间或偏移量对消息进行追溯。
- 记录流媒体的中心。
- 大数据集成。
- 可靠的 FIFO 和严格的有序消息传递在同一队列中。
- 高效的推拉消费模式。
- 单个队列中的百万级消息累积容量。
- 多种消息传递协议，如 JMS 和 OpenMessaging。
- 灵活的分布式横向扩展部署架构。
- Lightning-fast 批处理消息交换系统。
- 各种消息过滤器机制，如 SQL 和 Tag。
- Docker 图像用于隔离测试和云隔离集群。
- 功能丰富的管理仪表板，用于配置、指标和监控。

RocketMQ 由 4 部分组成，分别是 name servers、brokers、producers 和 consumers，如图 9-15 所示。

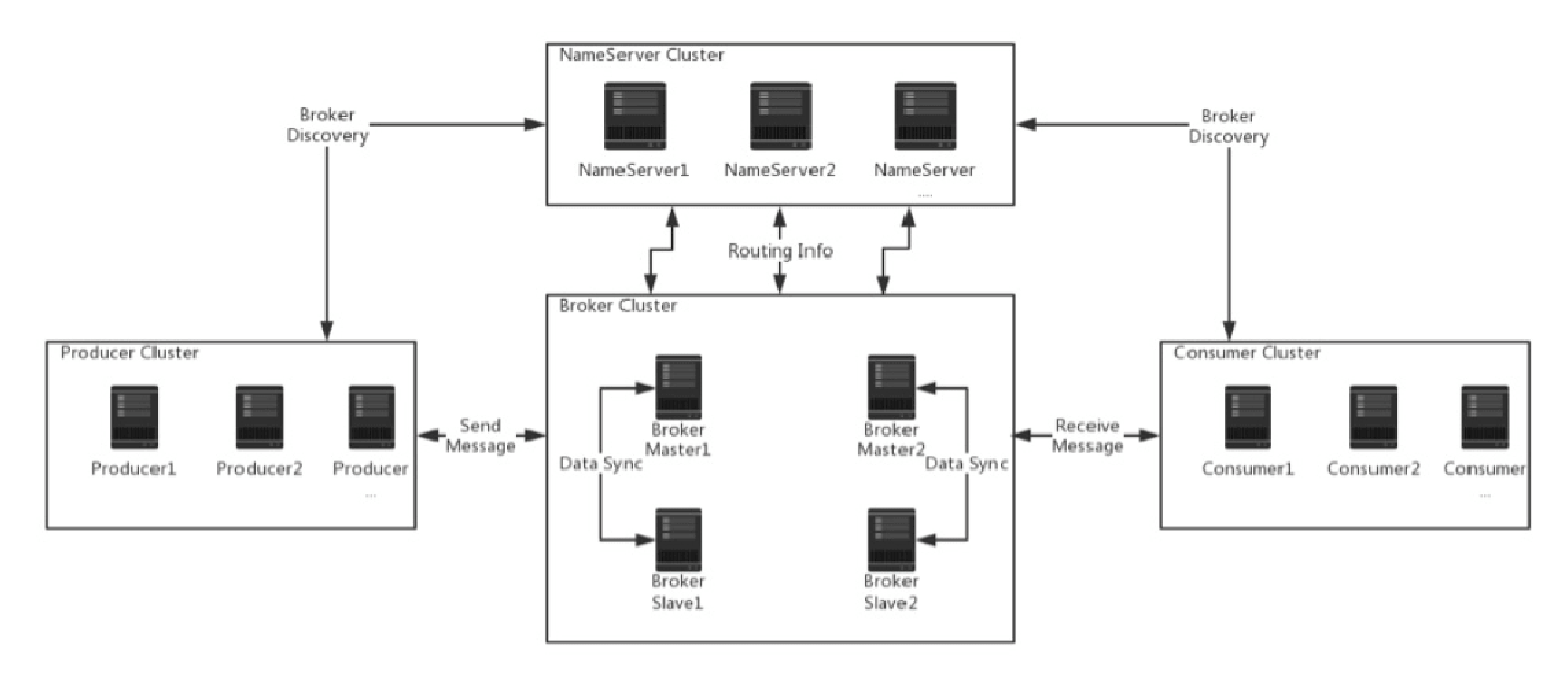


图 9-15　RocketMQ 组成图

9.3.2　Spring Boot 使用 RocketMQ

由于官网已经详细介绍了如何使用 RocketMQ，因此本小节仅以发送简单消息为例介绍 Spring Boot 如何使用 RocketMQ。

Spring Boot 使用 RocketMQ 消息队列大致分为三步：

（1）加入 RocketMQ 依赖。

（2）配置 RocketMQ 服务信息。

（3）编写生产者和消费者。

1. 加入 RocketMQ 依赖

新建项目，在 pom 文件中加入 RocketMQ 依赖，如代码清单 9-21 所示。

代码清单 9-21　RocketMQ 项目依赖代码

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.2.0</version>
</dependency>
```

2. 配置 RocketMQ 服务信息

在配置文件中加入 RocketMQ 服务的配置信息，如代码清单 9-22 所示。

代码清单 9-22 RocketMQ 项目配置文件

```
# 消费者的组名
apache.rocketmq.consumer.PushConsumer=PushConsumer

# 生产者的组名
apache.rocketmq.producer.producerGroup=Producer

# NameServer 地址
apache.rocketmq.namesrvAddr=localhost:9876
```

3. 编写生产者和消费者

这里以发送 10 条简单消息为例，创建一个生产者，这里使用的是默认生产者 DefaultMQProducer，在构建生产者的时候使用构造方法设置生产者的组名。使用 setNamesrvAddr() 方法设置 NameServer，如果有多个 NameServer，就使用逗号分隔。这里需要注意一点，生产者对象只调用一次 start 方法即可，不需要每次都调用。在构建消息体时设置 topic 和 tags。完整内容如代码清单 9-23 所示。

代码清单 9-23 RocketMQ 项目生产者代码

```
    @Component
    public class RocketMQSender {
        @Value("${apache.rocketmq.producer.producerGroup}")
        private String producerGroup;
        @Value("${apache.rocketmq.namesrvAddr}")
        private String namesrvAddr;
        private static final Logger log = LoggerFactory.getLogger
(RocketMQSender.class);

        public void defaultMQProducer() {
            DefaultMQProducer producer = new DefaultMQProducer(producerGroup);
            producer.setVipChannelEnabled(false);
            //指定 NameServer 地址，多个地址以 ; 隔开
            producer.setNamesrvAddr(namesrvAddr);
            try {
                producer.start();
                Message message = new Message("TopicTest", "push", "【发送消息】
".getBytes());
                StopWatch stop = new StopWatch();
                stop.start();

                for (int i = 0; i < 10; i++) {
```

```
                SendResult result = producer.send(message, new
MessageQueueSelector() {
                    @Override
                    public MessageQueue select(List<MessageQueue> mqs, Message
msg, Object arg) {
                        Integer id = (Integer) arg;
                        int index = id % mqs.size();
                        return mqs.get(index);
                    }
                },1);
                log.info("发送响应：MsgId:" + result.getMsgId() + "，发送状态:" +
result.getSendStatus());
            }
            stop.stop();
            log.info("----------------发送十条消息耗时：" +
stop.getTotalTimeMillis());
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            producer.shutdown();
        }
    }

}
```

接下来编写一个消费者，其中的设置与生产者类似，这里就不做过多介绍了。完整消费者如代码清单 9-24 所示。

代码清单 9-24　RocketMQ 项目消费者代码

```
@Component
public class RocketMQReceiver {
    @Value("${apache.rocketmq.consumer.PushConsumer}")
    private String consumerGroup;
    @Value("${apache.rocketmq.namesrvAddr}")
    private String namesrvAddr;
    private static final Logger log = LoggerFactory.getLogger
(RocketMQReceiver.class);

    //@PostContruct 是 spring 框架的注解，在方法上加该注解会在项目启动的时候执行该方法，
也可以理解为在 spring 容器初始化的时候执行该方法
    @PostConstruct
    public void defaultMQPushConsumer() {
        //消费者的组名
```

```
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer
(consumerGroup);
        //指定 NameServer 地址，多个地址以 ; 隔开
        consumer.setNamesrvAddr(namesrvAddr);
        try {
            //订阅 PushTopic 下 Tag 为 push 的消息
            consumer.subscribe("TopicTest", "push");
            //设置 Consumer 第一次启动是从队列头部开始消费还是从队列尾部开始消费
            //如果非第一次启动，那么按照上次消费的位置继续消费
            consumer.setConsumeFromWhere(ConsumeFromWhere.
CONSUME_FROM_FIRST_OFFSET);
            consumer.registerMessageListener((MessageListenerConcurrently)
(list, context) -> {
                try {
                    for (MessageExt messageExt : list) {
                         //输出消息内容
                        log.info("messageExt: " + messageExt);
                        String messageBody = new String(messageExt.getBody());
                        //输出消息内容
                        log.info("【消费响应】: msgId : " + messageExt.getMsgId()
+ ", msgBody : " + messageBody);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                    //稍后再试
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }
                //消费成功
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            });
            consumer.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //@PostContruct 是 spring 框架的注解，在方法上加该注解会在项目启动的时候执行该方法，
也可以理解为在 spring 容器初始化的时候执行该方法
    @PostConstruct
    public void defaultMQPushConsumer2() {
        //消费者的组名
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("aaa");
        //指定 NameServer 地址，多个地址以 ; 隔开
```

```
        consumer.setNamesrvAddr(namesrvAddr);
        try {
            //订阅 PushTopic 下 Tag 为 push 的消息
            consumer.subscribe("TopicTest", "push");
            //设置 Consumer 第一次启动是从队列头部开始消费还是从队列尾部开始消费
            //如果非第一次启动，那么按照上次消费的位置继续消费
            consumer.setConsumeFromWhere(ConsumeFromWhere.
CONSUME_FROM_FIRST_OFFSET);
            consumer.registerMessageListener((MessageListenerConcurrently)
(list, context) -> {
                try {
                    for (MessageExt messageExt : list) {
                        //输出消息内容
                        log.info("---- messageExt: " + messageExt);
                        String messageBody = new String(messageExt.getBody());
                        //输出消息内容
                        log.info("----【消费响应】: msgId : " +
messageExt.getMsgId() + ", msgBody : " + messageBody);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                    //稍后再试
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }
                //消费成功
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            });
            consumer.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

最后编写一个 RocketMQController 类来调用生产者发送消息，如代码清单 9-25 所示。

代码清单 9-25　RocketMQ 项目测试类代码

```
@RestController
public class RocketMQController {
    @Autowired
    private RocketMQSender rocketMQSender;

    @GetMapping("testRocketmq")
    public void testRocketmq(){
```

```
            rocketMQSender.defaultMQProducer();
        }
    }
```

启动项目后，在浏览器中访问 http://localhost:8080/testRocketmq，如图 9-16 所示。

```
essageThread_7] c.springboot.receiver.RocketMQReceiver   : messageExt: MessageExt [queueId=1, storeSize=180, queueOffset=67, sysFlag=0, bornTimestamp=1547972026921, b
essageThread_7] c.springboot.receiver.RocketMQReceiver   : 【消费响应】: msgId : A9FE2FFB03FE18B4AAC2655459F70000,  msgBody : 【发送消息】
io-8080-exec-3] com.springboot.sender.RocketMQSender     : 发送响应: MsgId:A9FE2FFB03FE18B4AAC2655459F70000, 发送状态:SEND_OK
essageThread_7] c.springboot.receiver.RocketMQReceiver   : ---- messageExt: MessageExt [queueId=1, storeSize=180, queueOffset=67, sysFlag=0, bornTimestamp=15479720269
essageThread_7] c.springboot.receiver.RocketMQReceiver   : ----【消费响应】: msgId : A9FE2FFB03FE18B4AAC2655459F70000,  msgBody : 【发送消息】
essageThread_8] c.springboot.receiver.RocketMQReceiver   : ---- messageExt: MessageExt [queueId=1, storeSize=180, queueOffset=68, sysFlag=0, bornTimestamp=15479720269
essageThread_8] c.springboot.receiver.RocketMQReceiver   : ----【消费响应】: msgId : A9FE2FFB03FE18B4AAC2655459F70000,  msgBody : 【发送消息】
io-8080-exec-3] com.springboot.sender.RocketMQSender     : 发送响应: MsgId:A9FE2FFB03FE18B4AAC2655459F70000, 发送状态:SEND_OK
essageThread_8] c.springboot.receiver.RocketMQReceiver   : messageExt: MessageExt [queueId=1, storeSize=180, queueOffset=68, sysFlag=0, bornTimestamp=1547972026935, b
essageThread_8] c.springboot.receiver.RocketMQReceiver   : 【消费响应】: msgId : A9FE2FFB03FE18B4AAC2655459F70000,  msgBody : 【发送消息】
io-8080-exec-3] com.springboot.sender.RocketMQSender     : 发送响应: MsgId:A9FE2FFB03FE18B4AAC2655459F70000, 发送状态:SEND_OK
io-8080-exec-3] com.springboot.sender.RocketMQSender     : ---------------发送十条消息耗时: 102
essageThread_9] c.springboot.receiver.RocketMQReceiver   : ---- messageExt: MessageExt [queueId=1, storeSize=180, queueOffset=69, sysFlag=0, bornTimestamp=15479720269
essageThread_9] c.springboot.receiver.RocketMQReceiver   : ----【消费响应】: msgId : A9FE2FFB03FE18B4AAC2655459F70000,  msgBody : 【发送消息】
essageThread_9] c.springboot.receiver.RocketMQReceiver   : messageExt: MessageExt [queueId=1, storeSize=180, queueOffset=69, sysFlag=0, bornTimestamp=1547972026954, b
essageThread_9] c.springboot.receiver.RocketMQReceiver   : 【消费响应】: msgId : A9FE2FFB03FE18B4AAC2655459F70000,  msgBody : 【发送消息】
ssageThread_10] c.springboot.receiver.RocketMQReceiver   : messageExt: MessageExt [queueId=1, storeSize=180, queueOffset=70, sysFlag=0, bornTimestamp=1547972026960, b
ssageThread_10] c.springboot.receiver.RocketMQReceiver   : 【消费响应】: msgId : A9FE2FFB03FE18B4AAC2655459F70000,  msgBody : 【发送消息】
ssageThread_10] c.springboot.receiver.RocketMQReceiver   : ---- messageExt: MessageExt [queueId=1, storeSize=180, queueOffset=70, sysFlag=0, bornTimestamp=15479720269
ssageThread_10] c.springboot.receiver.RocketMQReceiver   : ----【消费响应】: msgId : A9FE2FFB03FE18B4AAC2655459F70000,  msgBody : 【发送消息】
```

图 9-16　RocketMQ 项目控制台输出

RocketMQ 消息发送已经成功了。还有很多特性，具体可以查看官网教程学习。

9.4　消息队列对比

前面学习了常用的几个消息队列，本节从以下几方面对常用消息队列进行比较。

（1）关注度

首先我们从百度搜索指数来看本章介绍的 3 个消息队列的关注情况，如图 9-17 所示。

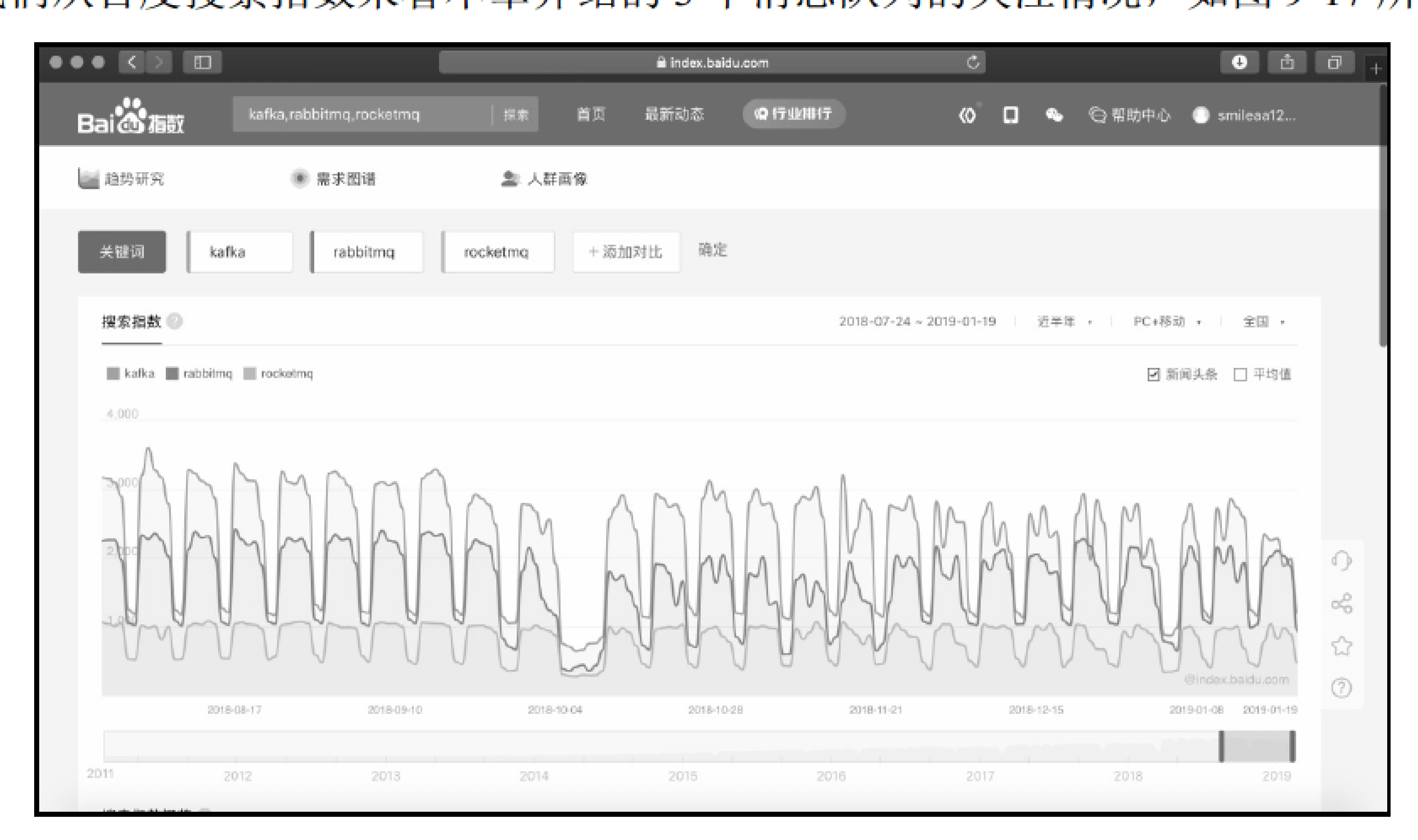


图 9-17　百度搜索指数队列搜索对比

从图 9-17 中可以看到，Kafka 消息队列远远领先于 RabbitMQ 消息队列，RocketMQ 最末。毕竟 RocketMQ 捐献给 Apache 基金会没有多久，这些老牌消息队列的关注度还是很高的。

接下来，我们来看谷歌趋势中的搜索指数对比，如图 9-18 所示。

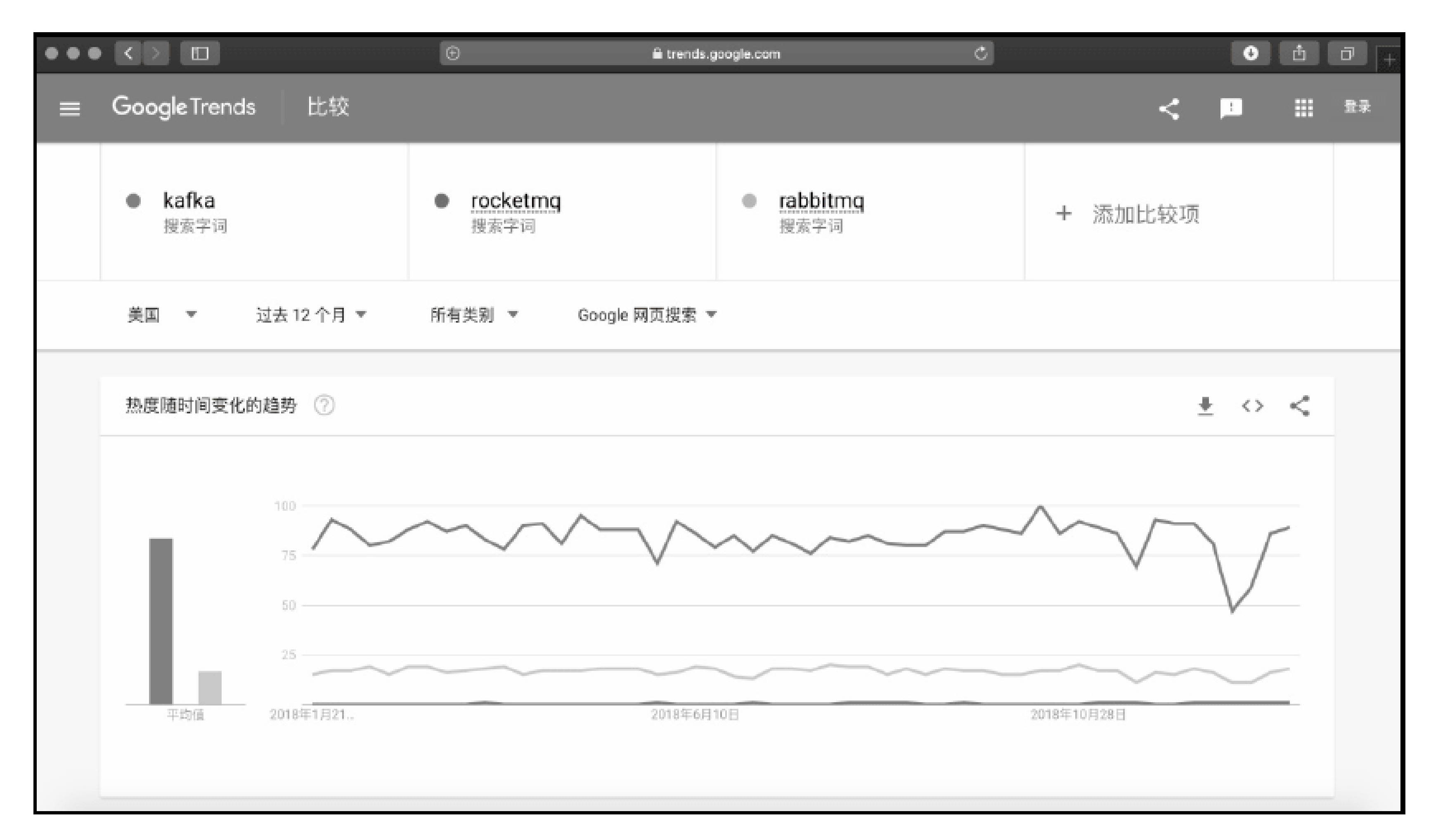


图 9-18　谷歌趋势消息队列搜索对比

与百度搜索相近，在关注度方面，Kafka 最高，RabbitMQ 其次，RocketMQ 最末。

（2）成熟度

从成熟度来看，其实对 RocketMQ 并不是十分公平。毕竟和老牌消息队列相比，它还是一个初出茅庐的新手。所以就成熟度来说，Kafka 和 RabbitMQ 已经是成熟消息队列，而 RocketMQ 属于比较成熟的消息队列，毕竟自从开源给 Apache 基金会，已经发布几个版本了。

（3）社区活跃度

社区活跃度方面，虽然国人对 RocketMQ 的呼声很高，并且已经有大部分企业准备使用 RocketMQ，但是还是无法与老牌的消息队列社区比较。

（4）吞吐量

吞吐量方面，查看阿里中间件博客对三者的对比，结果如图 9-19 所示。

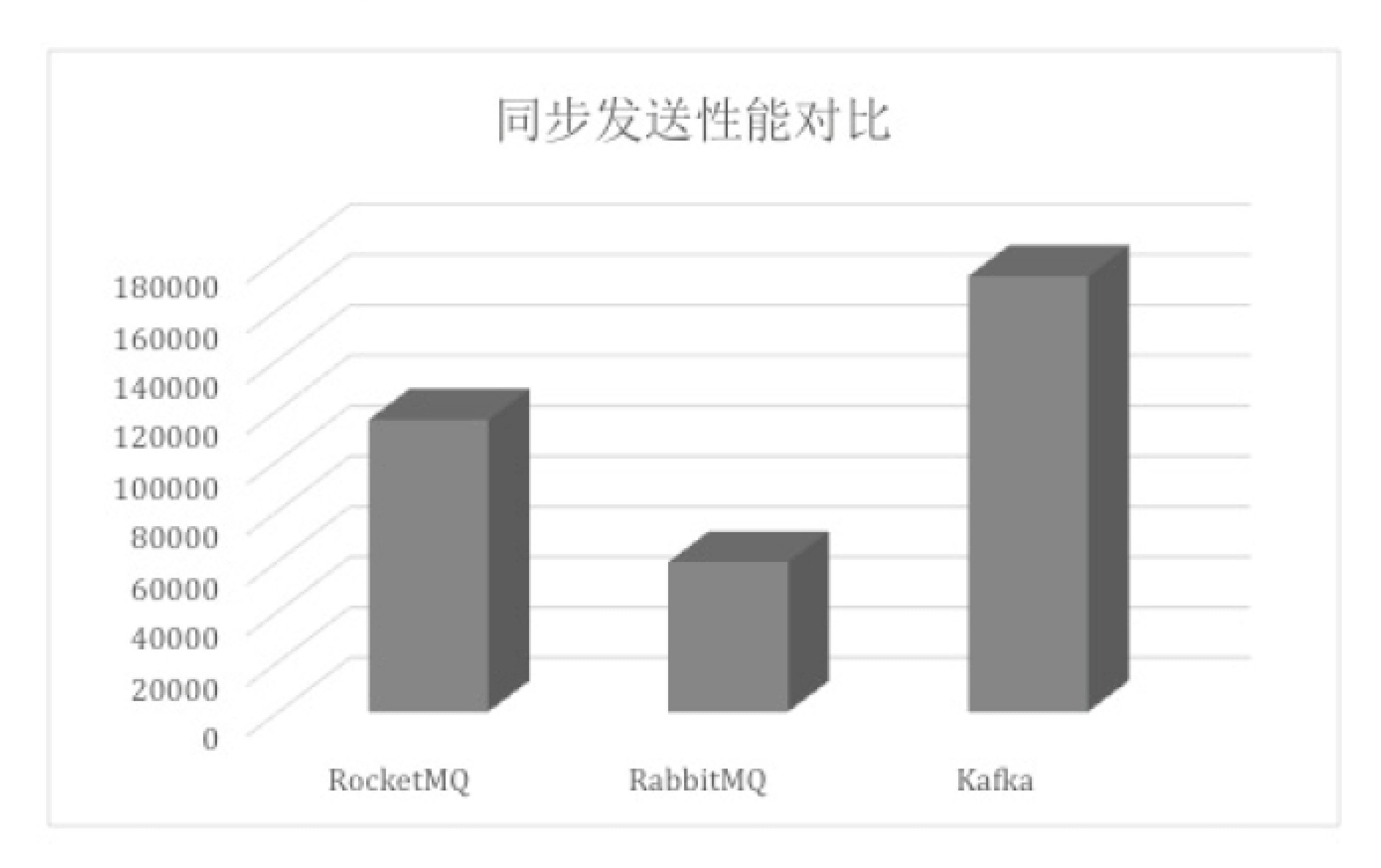


图 9-19　消息队列对比图

在同步发送场景中，3 个消息中间件的表现区别明显：

- Kafka 的吞吐量高达 17.3w/s，不愧是高吞吐量消息中间件的行业老大。这主要取决于它的队

列模式保证了写磁盘的过程是线性 IO，此时 broker 磁盘 IO 已达瓶颈。

- RocketMQ 也表现不俗，吞吐量为 11.6w/s，磁盘 IO %util 已接近 100%。RocketMQ 的消息写入内存后即返回 ack，由单独的线程专门做刷盘的操作，所有的消息均是顺序写文件的。
- RabbitMQ 的吞吐量为 5.95w/s，CPU 资源消耗较高。它支持 AMQP 协议，实现非常重量级，为了保证消息的可靠性，在吞吐量方面做了取舍。我们还做了 RabbitMQ 在消息持久化场景下的性能测试，吞吐量在 2.6w/s 左右。

测试结论：在服务端处理同步发送的性能上，Kafka>RocketMQ>RabbitMQ。

（5）可靠性

可靠性方面，RabbitMQ 最好；其次是 RocketMQ；Kafka 最差，会有丢失消息的情况。

（6）事务

在事务方面，只有 RocketMQ 提供了事务支持，其他消息服务都不提供事务支持。

（7）个人建议

如果现有消息队列用得很好，没有特别的性能要求，不需要重复造轮子。另外，要结合现有场景来使用，不要盲目追求吞吐量等指标。

9.5 小　结

本章对 Spring Boot 使用消息队列进行了介绍，包括传统流行的 RabbitMQ 消息队列、Kafka 消息队列以及阿里巴巴经过多次“双 11”测试的 RocketMQ 消息队列，最后对消息队列进行了对比。相信经过学习，读者会对使用消息队列有所了解，从而在今后的工作中使用消息队列时不再迷茫，找到正确的方向。

第 10 章

Spring Boot 的搜索之旅

当我们在访问购物网站的时候（比如淘宝、京东），根据意愿输入任意关键字，就可以查询出与关键字相关的内容，实现这项功能是怎么做到的呢？通常这项功能是通过全文检索来实现的。而对于开源常用的全文检索工具，基本上大多数企业都会选用 Apache Solr 或者 Elasticsearch。本章将带领大家学习 Spring Boot 对二者的使用。

10.1 使用 Solr

10.1.1 Solr 简介

Solr（官网地址：http://lucene.apache.org/solr/）是基于 Apache Lucene 构建的流行、快速、开源的企业搜索平台。Solr 是一个独立的企业搜索服务器，具有类似 REST 的 API。Solr 支持多种类型文件创建索引，可以通过 JSON、XML、CSV 或二进制文件将文档放入其中。可以通过 HTTP GET 查询它并接收 JSON、XML、CSV 或二进制结果。

Solr 是 Apache 基金会的项目，在官网上可以查看最新版本，如图 10-1 所示。

Solr 是一个高性能的搜索引擎，其采用 Java 5 开发，正是由于它基于 Lucene 的全文搜索服务器，因此提供了比 Lucene 更为丰富的查询语言，同时实现了可配置、可扩展并对查询性能进行了优化，并且提供了一个完善的功能管理界面，是一款非常优秀的全文搜索引擎。

通常来说，Solr 运行在 Servlet 容器（如 Tomcat、Jetty、WebLogic）中，提供了独立的管理平台，便于开发者使用及管理。

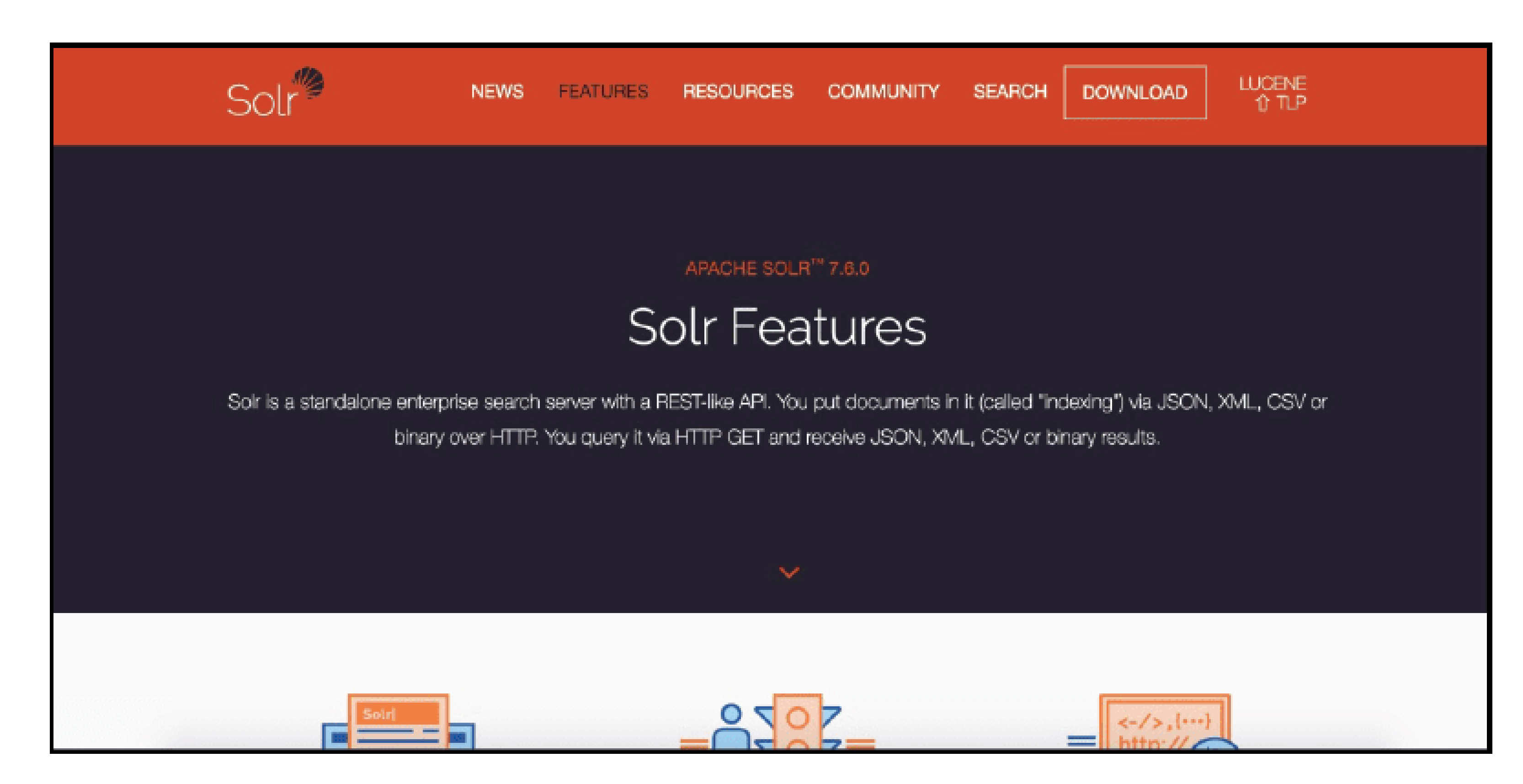


图 10-1 Solr 官网

10.1.2 Spring Boot 使用 Solr

Spring Boot 对于 Solr 的使用其实是很简单的，在 Spring Boot 的 starter 中就有 Solr 的依赖，直接引入即可，如代码清单 10-1 所示。

代码清单 10-1 Solr 项目依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-solr</artifactId>
</dependency>
```

接下来，需要在配置文件中配置 Solr 服务器相关的信息，如代码清单 10-2 所示。

代码清单 10-2 Solr 项目配置文件代码

```
## solr 服务器地址，可以在后面直接定义索引名称，如 http://ip:端口/solr/test_solr
spring.data.solr.host=http://ip:端口/solr
## solr 索引名称（这个是自定义的属性，方便使用）
spring.data.solr.collectionName=test_solr
```

本小节只是对 Spring Boot 结合 Solr 的使用进行介绍，所以没有创建过多的类。接下来创建一个实体类 Goods，如代码清单 10-3 所示。

代码清单 10-3 Solr 项目 Goods 实体类代码

```
public class Goods {
    private Long id;
    private String goodsName;
    private String goodsIntroduce;
    private Double goodsPrice;
```

```
    ... //这里省略 set、get、构造函数、toString 等方法
}
```

在 Spring Boot 操作 Solr 的时候，所有操作都是通过 SolrClient 进行的。接下来我们创建一个 SolrController 进行操作。在类中注入 SolrClient（以下简称 client），并且将前面介绍的索引名称自定义属性定义进来，如代码清单 10-4 所示。

代码清单 10-4　SolrController 类基础配置信息

```
@RestController
public class SolrController {

    @Autowired
    private SolrClient client;

    @Value("${spring.data.solr.collectionName}")
    private String collectionName;

}
```

Controller 基本上定义好了。接下来从 5 个方面对 Solr 的使用进行学习。

1. 保存或修改

在保存或修改中，开始对传入 Goods 对象的 Id 进行判断，如果不存在，就获取当前系统时间戳作为 Id，做保存操作；如果存在，就做修改操作。保存和修改都是通过 client.add()进行操作的，在 Solr 中会判断 Id 是否存在，若存在，则修改；若不存在，则新增。对于新增、修改或删除操作，需要在行为后使用 client.commit()方法。在 commit 方法中传入操作索引名称提交操作，如代码清单 10-5 所示。

代码清单 10-5　保存或修改方法代码

```
    @PostMapping("saveOrUpdate")
    public String saveOrUpdate(@RequestBody Goods goods) {
        if(goods.getId() == null){
            goods.setId(System.currentTimeMillis());
        }
        try {
            SolrInputDocument solrInputDocument = new SolrInputDocument();
            solrInputDocument.setField("id", goods.getId());
            solrInputDocument.setField("goodsName", goods.getGoodsName());
            solrInputDocument.setField("goodsIntroduce",
goods.getGoodsIntroduce());
            solrInputDocument.setField("goodsPrice", goods.getGoodsPrice());
            client.add(collectionName, solrInputDocument);
```

```
            client.commit(collectionName);

            return goods.toString();

        } catch (Exception e) {
            e.printStackTrace();
        }
        return "error";
    }
```

2. 删除

删除比较简单，调用 client.deleteById()方法对指定 Id 的索引进行删除，如代码清单 10-6 所示。

代码清单 10-6　删除方法代码

```
    @GetMapping("delete")
    public String delete(String id)  {
        try {
            client.deleteById(collectionName,id);
            client.commit(collectionName);
            return id;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "error";
    }
```

3. 删除所有

删除所有是删除的改进版，使用 Solr 的通配符，如代码清单 10-7 所示。

代码清单 10-7　删除所有方法代码

```
    @GetMapping("deleteAll")
    public String deleteAll(){
        try {
            client.deleteByQuery(collectionName,"*:*");
            client.commit(collectionName);
            return "success";
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "error";
    }
```

4. 根据 Id 查询

根据 Id 查询方法调用了 client.getById()方法，在方法内传入索引名称和 Id，如代码清单 10-8

所示。

代码清单 10-8 根据 Id 查询代码

```
@GetMapping("getGoodsById")
public String getGoodsById(String id) throws Exception {
    SolrDocument document = client.getById(collectionName, id);
    return document.toString();
}
```

5. 复杂查询

复杂查询是使用 Solr 的重点，在讲解这个方法之前，先查看如图 10-2 所示的页面。



图 10-2 代码清单 10-2 Solr 管理页面

图 10-2 中的参数说明如下。

1. 查询条件介绍

- q：查询的关键字，默认值为*:*，代表查询所有，也可以指定条件，比如 id:1。
- fq：过虑查询，提供一个可选的筛选器查询。返回在 q 查询结果中，同时符合 fq 筛选条件的结果，例如 q=id:1&fq=goodsPrice:[90 TO 100]，找关键字 id 为 1 并且 goodsPrice 在 90 ~ 100 之间的结果。
- sort：排序方式，例如 sort 的值为 id desc，表示按照 id 降序排序。
- start：返回结果的第几条记录开始，一般分页用，默认从 0 开始。
- rows：指定返回结果最多有多少条记录，默认值为 10，配合 start 实现分页。
- fl：指定返回字段，用逗号或空格分隔。注意字段区分大小写，例如，fl 的值为 id,goodsName。
- df：默认的查询字段，一般需要开发人员指定。
- wt：指定输出格式，有 JSON、XML、Python、Ruby、PHP、CSV。
- indent off：返回的结果是否缩进，默认关闭，用 indent=true|on 开启，一般调试 JSON、PHP、

PHPS、Ruby 输出才有必要用这个参数。

- debugQuery：设置返回结果是否显示 Debug 信息。

2. Solr 的检索运算符

- “:” 指定字段查指定值，如返回所有值*:*。
- “?” 表示单个任意字符的通配，比如硬毛的?刷可以匹配硬毛的牙刷。
- “” 表示多个任意字符的通配，比如硬毛刷可以匹配硬毛的牙刷（注意：不能在检索的项开始使用*或者?符号）。
- “~” 表示模糊检索，比如我们想要检索到拼写类似于 administrator 的项这样写：administrator~，将找到形如 administerater 和 administratior 的单词。同时，模糊检索还支持设置相似度，比如我们想要检索到拼写类似于 admin 的同时相似度 0.8 以上的项可以这样写：admin~ 0.8，将返回相似度在 0.8 以上的记录。
- AND、|| 布尔操作符。
- OR、&& 布尔操作符。
- NOT、!、- 排除操作符，不能单独与项使用构成查询。
- “+” 存在操作符，要求符号“+”后的项必须在文档相应的域中存在。
- ( ) 用于构成子查询。
- [] 包含范围检索，如检索某时间段的记录，包含头尾，goodsPrice:[90 TO 100]。
- {} 不包含范围检索，如检索某时间段的记录，不包含头尾，goodsPrice:{90 TO 100}。

到这里，我们对 Solr 的使用已经有了大致的了解，这些功能已经可以满足我们日常的使用。接下来介绍复杂查询，如代码清单 10-9 所示。

代码清单 10-9 复杂查询方法代码

```
@GetMapping("search")
public Map<String, Object> search(String keyword){
    //返回集合
    Map<String,Object> returnMap = new HashMap();
    try {
        SolrQuery params = new SolrQuery();
        //查询条件
        params.set("q", keyword);
        //过滤条件
        params.set("fq", "goodsPrice:[100 TO 100000]");
        //排序
        params.addSort("id", SolrQuery.ORDER.asc);
        //分页
        //从第几条记录开始
        params.setStart(0);
        //最多返回多少条记录
        params.setRows(20);
        //默认域
```

```
            params.set("df", "goodsIntroduce");
            //只查询指定域
            params.set("fl", "id,goodsName,goodsIntroduce,goodsPrice");
            //高亮
            //打开开关
            params.setHighlight(true);
            //指定高亮域
            params.addHighlightField("goodsIntroduce");
            params.addHighlightField("goodsName");
            //设置高亮前缀
            params.setHighlightSimplePre("<span style='color:red'>");
            //设置高亮后缀
            params.setHighlightSimplePost("</span>");
            QueryResponse queryResponse = client.query(collectionName,
params);
            SolrDocumentList results = queryResponse.getResults();
            //返回行数
            long numFound = results.getNumFound();
            //获取高亮显示的结果，高亮显示的结果和查询结果是分开放的
            Map<String, Map<String, List<String>>> highlight = queryResponse.
getHighlighting();
            results.forEach(result->{
                Map map = highlight.get(result.get("id"));
                result.addField("goodsNameHH",map.get("goodsName"));
                result.addField("goodsIntroduceHH",
map.get("goodsIntroduce"));
            });
            returnMap.put("numFound", numFound);
            returnMap.put("results", results);
            return returnMap;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
```

代码清单 10-9 对前面介绍的查询条件进行了一定的使用，需要注意这里使用 addHighlightField 对高亮字段进行定义，对高亮字段前后都加入 span 标签并且样式设置为红色，当然也可以根据自己的需求进行修改。

Spring Boot 对 Solr 的操作到这里基本上就结束了。由于篇幅所限，就不给出相关测试了，感兴趣的读者可以自行测试。

10.2 使用 Elasticsearch

10.2.1 Elasticsearch 简介

Elasticsearch（官网地址：https://www.elastic.co/cn/products/elasticsearch）是一个分布式可扩展的实时搜索和分析引擎。Elasticsearch 由 Java 开发，其底层基于 Lucene 的搜索服务器，对外提供了 RESTful Web 接口。

在美国时间 2018 年 10 月 5 日，Elasticsearch 在美国纽约证券交易所上市，在上市当天，Elastic 公司在官网发布了公开信，如图 10-3 所示。

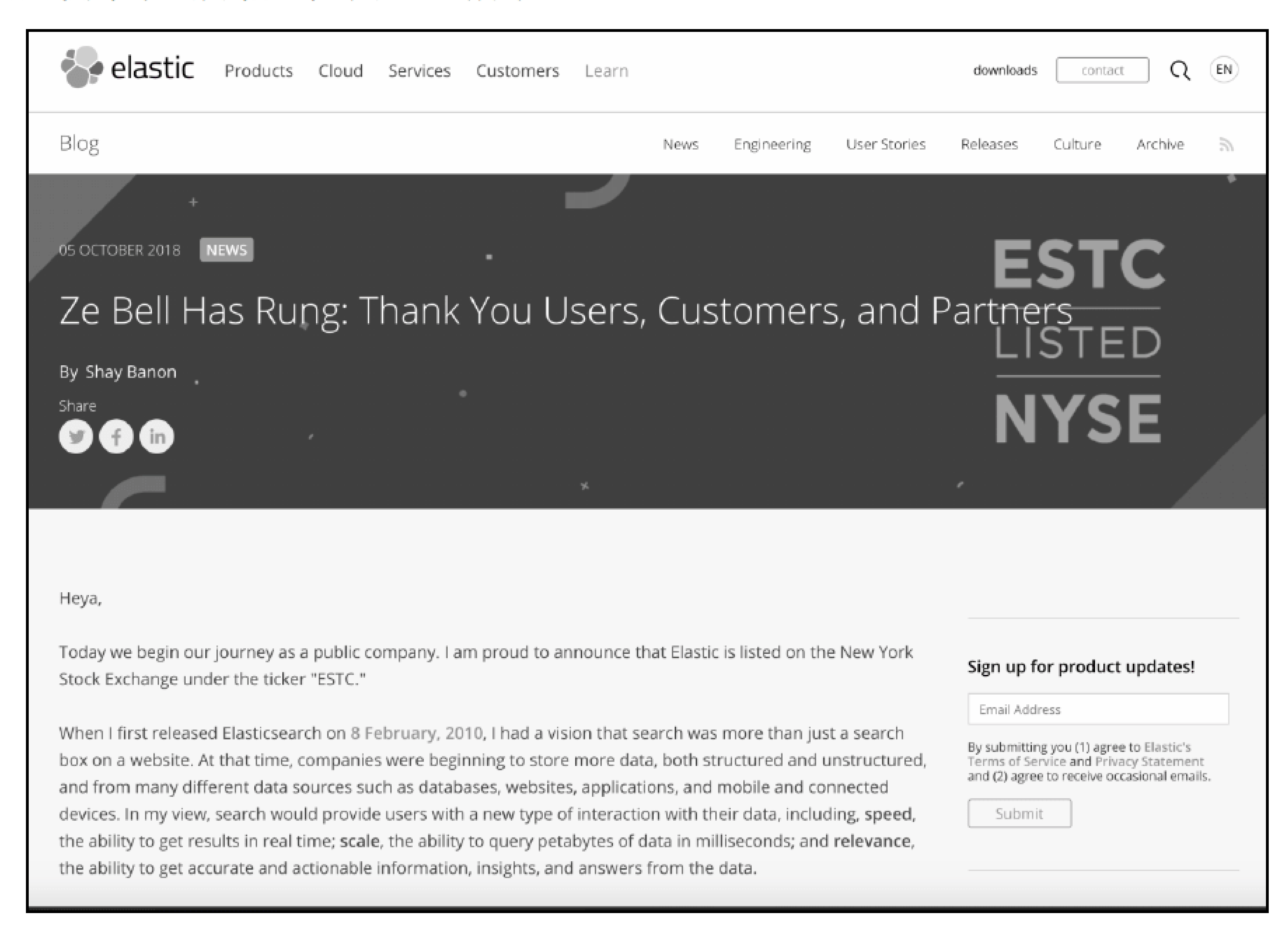


图 10-3　Elastic 官网公开信

Elastic 公司成立于 2012 年，正是由于 Elasticsearch 而闻名，该公司还有很多优秀的产品，如分布式日志解决方案 ELK（Elasticsearch、Logstash、Kibana）、Beats 等。

目前，Elasticsearch 被很多大型组织使用，比如我们常用的代码托管网站 Github，笔者认为有效解决问题的 IT 问答网站 Stack Overflow 以及维基百科都在使用它。在国内，许多电商公司的商品搜索和管理使用得很多，Elasticsearch 已经成为很多公司解决搜索问题的必备良药。

10.2.2 Spring Boot 使用 Elasticsearch

Spring Boot 提供了使用 Elasticsearch 的 starter。新建项目，在项目中加入 Elasticsearch 依赖，如代码清单 10-10 所示。

代码清单 10-10　Elasticsearch 项目依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
```

接下来，在配置文件中配置 Elasticsearch 服务器地址。由于样例 demo 使用的 Elasticsearch 是单节点的，因此只需要配置 Elasticsearch 地址即可，如代码清单 10-11 所示。

代码清单 10-11　Elasticsearch 项目配置文件

```
spring.data.elasticsearch.cluster-nodes = 127.0.0.1:9300
```

10.2.3　使用 Elasticsearch Repository 进行操作

Elsticsearch 提供了像 JPA 一样操作的 Elasticsearch Repository，其操作与 JPA 使用类似，这里展示几个简单的使用方法，如代码清单 10-12 所示。

代码清单 10-12　BaseOperationController 类代码

```
@RestController
public class BaseOperationController {

    @Autowired
    private ArticleRepository articleRepository;

    @PostMapping("saveOrUpdate")
    public String saveOrUpdate(@RequestBody Article article){
        if(article.getId()==null){
            article.setId(System.currentTimeMillis());
        }
        articleRepository.save(article);
        return "保存成功";
    }

    @GetMapping("delete")
    public String delete(Long id){
        articleRepository.deleteById(id);
        return "删除成功";
    }

    @GetMapping("findById")
    public Article findById(long id){
        Article article = articleRepository.findById(id).get();
        return article;
    }
```

```
    @GetMapping("findAll")
    public Iterable<Article> findAll(){
        return articleRepository.findAll();
    }
}
```

是不是和 JPA 很相似？感兴趣的读者可以研究一下。

10.2.4 使用 Elasticsearch Template 进行操作

Spring Boot 的 spring-boot-starter-data-elasticsearch 还提供了 Elasticsearch Template，这个模板类可以进行查询之类的操作，还可以进行一些配置相关的操作，如创建和删除索引、设置 Mapping 等。这里分别展示创建索引、删除索引、判断是否存在此索引、判断索引中是否存在当前 type、获取 Mapping 和获取 Setting 操作，如代码清单 10-13 所示。

代码清单 10-13 ElasticOperationControler 类代码

```
@RestController
public class ElasticOperationControler {

    @Autowired
    private ElasticsearchTemplate elasticsearchTemplate;

    @GetMapping("createIndex")
    public boolean createIndex(String indexName){
        return elasticsearchTemplate.createIndex(indexName);
    }

    @GetMapping("deleteIndex")
    public boolean deleteIndex(String indexName){
        return elasticsearchTemplate.deleteIndex(indexName);
    }

    @GetMapping("indexIsExist")
    public boolean indexIsExist(String indexName){
        return elasticsearchTemplate.indexExists(indexName);
    }

    @GetMapping("typeIsExist")
    public boolean typeIsExist(String indexName,String type){
        return elasticsearchTemplate.typeExists(indexName,type);
    }

    @GetMapping("getMapping")
    public Map getMapping(String indexName, String type){
        return elasticsearchTemplate.getMapping(indexName,type);
```

```
    }

    @GetMapping("getSetting")
    public Map getSetting(String indexName){
        return elasticsearchTemplate.getSetting(indexName);
    }
}
```

上面的内容都比较简单，所以笔者只是一笔带过，接下来是本节的重点内容。

10.2.5　非聚合查询

本小节先带领大家熟悉一下 Elasticsearch 的几种操作，然后进行几个查询来总结学习的内容。

非聚合查询大致分为 6 种：单匹配查询、多匹配查询、全匹配查询、模糊查询、范围查询和组合查询。

1. 单匹配查询

在单匹配查询中，分为两种场景，分别是使用分词器和不使用分词器（分词器是将用户输入的一段文本分隔成符合逻辑的多个词语，后续会专门讲解）。设置单匹配不分词查询条件如代码清单 10-14 所示。

代码清单 10-14　使用 termQuery 创建 QueryBuilder 代码

```
QueryBuilder queryBuilder=QueryBuilders.termQuery("fieldName",
"fieldValue");
```

设置单匹配分词查询条件如代码清单 10-15 所示。

代码清单 10-15　使用 matchQuery 创建 QueryBuilder 代码

```
QueryBuilder queryBuilder = QueryBuilders.matchQuery("fieldName",
"fieldValue");
```

例如，QueryBuilder queryBuilder=QueryBuilders.termQuery("articleName", "test"); 是指在 articleName 中包含 test 即可被查询到。

2. 多匹配查询

多匹配查询与单匹配查询的使用方式类似，包含使用分词器和不使用分词器。设置多匹配不分词查询条件如代码清单 10-16 所示。

代码清单 10-16　使用 termsQuery 创建 QueryBuilder 代码

```
QueryBuilder queryBuilder=QueryBuilders.termsQuery("fieldName",
"fieldlValue1","fieldlValue2...");
```

设置多匹配分词查询条件如代码清单 10-17 所示。

代码清单 10-17　使用 multiMatchQuery 创建 QueryBuilder 代码

```
    QueryBuilder queryBuilder= QueryBuilders.multiMatchQuery("fieldlValue",
"fieldName1", "fieldName2", "fieldName3");
```

例如，QueryBuilder queryBuilder=QueryBuilders.termsQuery("articleName", "test","article");是指在 articleName 中既包含 test 又包含 article 才会被查询到。

3. 全匹配查询

全匹配查询属于多匹配查询的特殊情况，通俗地理解，就是没有设置任何查询条件。这里将这种情况与多匹配查询分开来讲，一般设置全匹配查询条件如代码清单 10-18 所示。

代码清单 10-18　使用 matchAllQuery 创建 QueryBuilder 代码

```
    QueryBuilder queryBuilder=QueryBuilders.matchAllQuery();
```

4. 模糊查询

模糊查询其实和 SQL 中的模糊很类似，大致分为如下几种：

（1）左右模糊查询：比如 QueryBuilders.queryStringQuery("fieldValue").field("fieldName");。

（2）相似内容的查询：如果不指定 filedName，就默认为全部，常用在相似内容的推荐上，比如 QueryBuilders.moreLikeThisQuery(new String[] {"fieldName"}).addLikeText("fieldValue");。

（3）前缀查询：如果字段没分词，就匹配整个字段前缀，比如 QueryBuilders.prefixQuery("fieldName","fieldValue"); 。

（4）分词模糊查询：通过增加 fuzziness 模糊属性来查询，如能够匹配 hotelName 为 tel 前或后加一个字母的文档，fuzziness 的含义是检索的 term 前后增加或减少 n 个单词的匹配查询，比如 QueryBuilders.fuzzyQuery("fieldName", "fieldValue").fuzziness(Fuzziness.ONE); 。

（5）通配符查询：支持*任意字符串，任意一个字符，通常使用 fieldValue 后拼接通配符，比如 QueryBuilders.wildcardQuery("fieldName","con*");。

5. 范围查询

顾名思义，范围查询就是给定范围区间查询，分为以下几种：

- 闭区间查询，如 QueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").from("fieldValue1").to("fieldValue2"); 。
- 开区间查询，如 QueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").from("fieldValue1").to("fieldValue2").includeUpper(false).includeLower(false);，默认值为 true，表示包含边界值。
- 大于查询，如 QueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").gt("fieldValue");。

- 大于等于查询，如 QueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").gte("fieldValue");。
- 小于查询，如 QueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").lt("fieldValue");。
- 小于等于查询，如 QueryBuilder queryBuilder = QueryBuilders.rangeQuery("fieldName").lte("fieldValue");。

6. 组合查询

组合查询其实就是将上述几种场景组合起来，与 SQL 中的查询条件一样，分为如下三种：

- 文档必须完全匹配条件，相当于 SQL 中的 AND 条件，如 QueryBuilders.boolQuery().must();。
- 文档必须完全不匹配条件，相当于 SQL 中的 NOT 条件，如 QueryBuilders.boolQuery().mustNot();。
- 文档匹配条件，多个条件满足一个即可，相当于 SQL 中的 OR 条件，如 QueryBuilders.boolQuery().should();。

10.2.6　聚合查询

（1）统计某个字段的数量

```
ValueCountBuilder vcb= AggregationBuilders.count("count_uid").field("uid");
```

（2）去重统计某个字段的数量（有少量误差）

```
CardinalityBuilder cb=
AggregationBuilders.cardinality("distinct_count_uid").field("uid");
```

（3）聚合过滤

```
FilterAggregationBuilder fab= AggregationBuilders.filter("uid_filter").
filter(QueryBuilders.queryStringQuery("uid:001"));
```

（4）按某个字段分组

```
TermsBuilder tb= AggregationBuilders.terms("group_name").field("name");
```

（5）求和

```
SumBuilder sumBuilder= AggregationBuilders.sum("sum_price").field("price");
```

（6）求平均

```
AvgBuilder ab= AggregationBuilders.avg("avg_price").field("price");
```

（7）求最大值

```
MaxBuilder mb= AggregationBuilders.max("max_price").field("price");
```

（8）求最小值

```
MinBuilder min= AggregationBuilders.min("min_price").field("price");
```

（9）按日期间隔分组

```
DateHistogramBuilder dhb= AggregationBuilders.dateHistogram("dh").
field("date");
```

（10）获取聚合里面的结果

```
TopHitsBuilder thb= AggregationBuilders.topHits("top_result");
```

（11）嵌套的聚合

```
NestedBuilder nb= AggregationBuilders.nested("negsted_path").path("quests");
```

（12）反转嵌套

```
AggregationBuilders.reverseNested("res_negsted").path("kps ");
```

10.2.7 复杂查询练习

结合案例内容，笔者准备了 5 个从简单到复杂的查询，分别说明如下：

场景一：不分词查询

查询 articleContent 带有“你好”或者 articleName 带有“你好”的文章列表，并且按照 readCount 倒叙排序，如代码清单 10-19 所示。

代码清单 10-19　场景一示例代码

```
    //http://localhost:8080/query10?keyword=你好
    @GetMapping("query1")
    public List<Article> query1(String keyword) {
        BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
        boolQueryBuilder.should(QueryBuilders.termQuery("articleContent",
keyword));
        boolQueryBuilder.should(QueryBuilders.termQuery("articleName",
keyword));
        FieldSortBuilder fieldSortBuilder = SortBuilders.fieldSort
("readCount").order(SortOrder.DESC);
        NativeSearchQueryBuilder nativeSearchQueryBuilder = new
NativeSearchQueryBuilder();
        nativeSearchQueryBuilder.withQuery(boolQueryBuilder);
        nativeSearchQueryBuilder.withSort(fieldSortBuilder);
        NativeSearchQuery nativeSearchQuery =
nativeSearchQueryBuilder.build();
        Page<Article> page = articleRepository.search(nativeSearchQuery);
```

```
        if (page != null) {
            return page.getContent();
        } else {
            return null;
        }
    }
```

场景二：不分词查询

查询 articleContent 带有“我们”或者“你好”并且 authorAge 在 20 岁以下的文章列表，并且按照 readCount 倒叙排序，如代码清单 10-20 所示。

代码清单 10-20　场景二示例代码

```
    //http://localhost:8080/query11?keyword=你好,我们
    @GetMapping("query2")
    public List<Article> query2(String... keyword) {
        BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
        boolQueryBuilder.must(QueryBuilders.termsQuery("articleContent",
keyword));
        boolQueryBuilder.must(QueryBuilders.rangeQuery
("authorAge").lt(20));
        FieldSortBuilder fieldSortBuilder = SortBuilders.fieldSort
("readCount").order(SortOrder.DESC);
        NativeSearchQueryBuilder nativeSearchQueryBuilder = new
NativeSearchQueryBuilder();
        nativeSearchQueryBuilder.withQuery(boolQueryBuilder);
        nativeSearchQueryBuilder.withSort(fieldSortBuilder);
        NativeSearchQuery nativeSearchQuery =
nativeSearchQueryBuilder.build();
        Page<Article> page = articleRepository.search(nativeSearchQuery);
        if (page != null) {
            return page.getContent();
        } else {
            return null;
        }
    }
```

场景三：分词查询

经过分词，查询 articleContent 带有“你好节日”一词分词后的文章列表，并且按照 authorAge 倒叙排序，如代码清代 10-21 所示。

代码清单 10-21　场景三示例代码

```
    //http://localhost:8080/query12?keyword=你好节日
```

```
        @GetMapping("query3")
        public List<Article> query3(String keyword) {
            BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
            boolQueryBuilder.should(QueryBuilders.matchQuery("articleContent",
keyword));
            FieldSortBuilder fieldSortBuilder =
SortBuilders.fieldSort("authorAge").order(SortOrder.DESC);
            NativeSearchQueryBuilder nativeSearchQueryBuilder = new
NativeSearchQueryBuilder();
            nativeSearchQueryBuilder.withQuery(boolQueryBuilder);
            nativeSearchQueryBuilder.withSort(fieldSortBuilder);
            NativeSearchQuery nativeSearchQuery =
nativeSearchQueryBuilder.build();
            Page<Article> page = articleRepository.search(nativeSearchQuery);
            if (page != null) {
                return page.getContent();
            } else {
                return null;
            }
        }
```

场景四：分页分词查询

经过分词，查询 articleContent 带有“你好节日”一词分词后的文章列表，并且按照 authorAge 倒叙排序，如代码清单 10-22 所示。

代码清单 10-22　场景四示例代码

```
        //http://localhost:8080/query13?keyword=你好节日&pageNum=1&pageSize=5
        @GetMapping("query4")
        public Page<Article> query4(String keyword, Integer pageNum, Integer
pageSize) {
            BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
            boolQueryBuilder.should(QueryBuilders.matchQuery("articleContent",
keyword));
            FieldSortBuilder fieldSortBuilder =
SortBuilders.fieldSort("authorAge").order(SortOrder.DESC);
            PageRequest pageRequest = new PageRequest(pageNum, pageSize);
            NativeSearchQueryBuilder nativeSearchQueryBuilder = new
NativeSearchQueryBuilder();
            nativeSearchQueryBuilder.withQuery(boolQueryBuilder);
            nativeSearchQueryBuilder.withSort(fieldSortBuilder);
            nativeSearchQueryBuilder.withPageable(pageRequest);
            NativeSearchQuery nativeSearchQuery = nativeSearchQueryBuilder.
build();
            Page<Article> page = articleRepository.search(nativeSearchQuery);
```

```
        if (page != null) {
            return page;
        } else {
            return null;
        }
    }
```

场景五：分页分词查询

这个场景需要提前安装分词器（这里使用的是 ik 分词器），经过分词，查询 articleContent 带有“你好节日”一词分词后的文章列表，并且按照 authorAge 倒叙排序，并且对匹配的词语设置高亮，如代码清单 10-23 所示。

代码清单 10-23　场景五示例代码

```
    //http://localhost:8080/query14?keyword=你好节日
&pageNum=1&pageSize=5&fieldNames=articleContent,articleName
    @GetMapping("query5")
    public Map<String, Object> query5(String keyword, Integer pageNum, Integer
pageSize, String... fieldNames) {
        //定义返回的 map
        Map<String, Object> returnMap = new HashMap<String,Object>();
        //构建请求构建器，设置查询索引
        SearchRequestBuilder builder = elasticsearchTemplate.
getClient().prepareSearch("testes");
        //构建查询构建器，设置分词器（如果没设置，就使用默认设置）
        QueryBuilder matchQuery = QueryBuilders.multiMatchQuery(keyword,
fieldNames).analyzer("ik_max_word");
        //构建高亮构建器
        HighlightBuilder highlightBuilder = new HighlightBuilder().
field("*").requireFieldMatch(false);
        highlightBuilder.preTags("<span style=\"color:red\">");
        highlightBuilder.postTags("</span>");
        //将高亮构建器、查询构建器、分页参数设置到请求构建器内
        builder.highlighter(highlightBuilder);
        builder.setQuery(matchQuery);
        builder.setFrom((pageNum - 1) * pageSize);
        builder.setSize(pageNum * pageSize);
        builder.setSize(pageSize);
        //执行搜索，返回搜索响应信息
        SearchResponse searchResponse = builder.get();
        SearchHits searchHits = searchResponse.getHits();
```

```
        //总命中数
        long total = searchHits.getTotalHits();
        returnMap.put("count", total);
        //将高亮字段封装到返回 map
        SearchHit[] hits = searchHits.getHits();
        List<Map<String,Object>> list = new ArrayList<>();
        Map<String,Object> map;
        for(SearchHit searchHit : hits){
            map = new HashMap<>();
            map.put("data",searchHit.getSourceAsMap());
            Map<String,Object> hitMap = new HashMap<>();
            searchHit.getHighlightFields().forEach((k,v) -> {
                String hight = "";
                for(Text text : v.getFragments()){
                    hight += text.string();
                }
                hitMap.put(v.getName(),hight);
            });
            map.put("highlight",hitMap);
            list.add(map);
        }
        returnMap.put("dataList", list);
        return returnMap;
    }
```

10.3 搜索引擎对比

前两节分别对当今开源流行的两大搜索引擎框架做了一定介绍，并且结合 Spring Boot 框架对二者进行了使用，究竟哪个框架更适合呢？结合笔者的经验，本节对二者进行比较。

10.3.1 技术背景

Solr 于 2006 年捐献给 Apache，成为 Apache 的孵化项目，一年后 Solr 成功孵化，发布了 1.2 版本，并且成为 Lucene 的子项目，从 1.4.x 版本以后，为了保持和 Lucene 同步，Solr 直接进入 3.0 版本。官网列出了 Solr 的版本（地址为 http://archive.apache.org/dist/lucene/solr/），如图 10-4 所示，在 Apache 开源基金会的强大背景下，Solr 的更新还是很频繁的。

Elasticsearch 是 2012 年 6 月开始开源的，对于 Elasticsearch 有这样一个故事：

伦敦的公寓内，Shay Banon 正在忙着寻找工作，而他的妻子正在蓝带（Le Cordon Bleu）烹饪学校学习厨艺。在空闲时间，他开始编写搜索引擎来帮助妻子管理越来越丰富的菜谱。

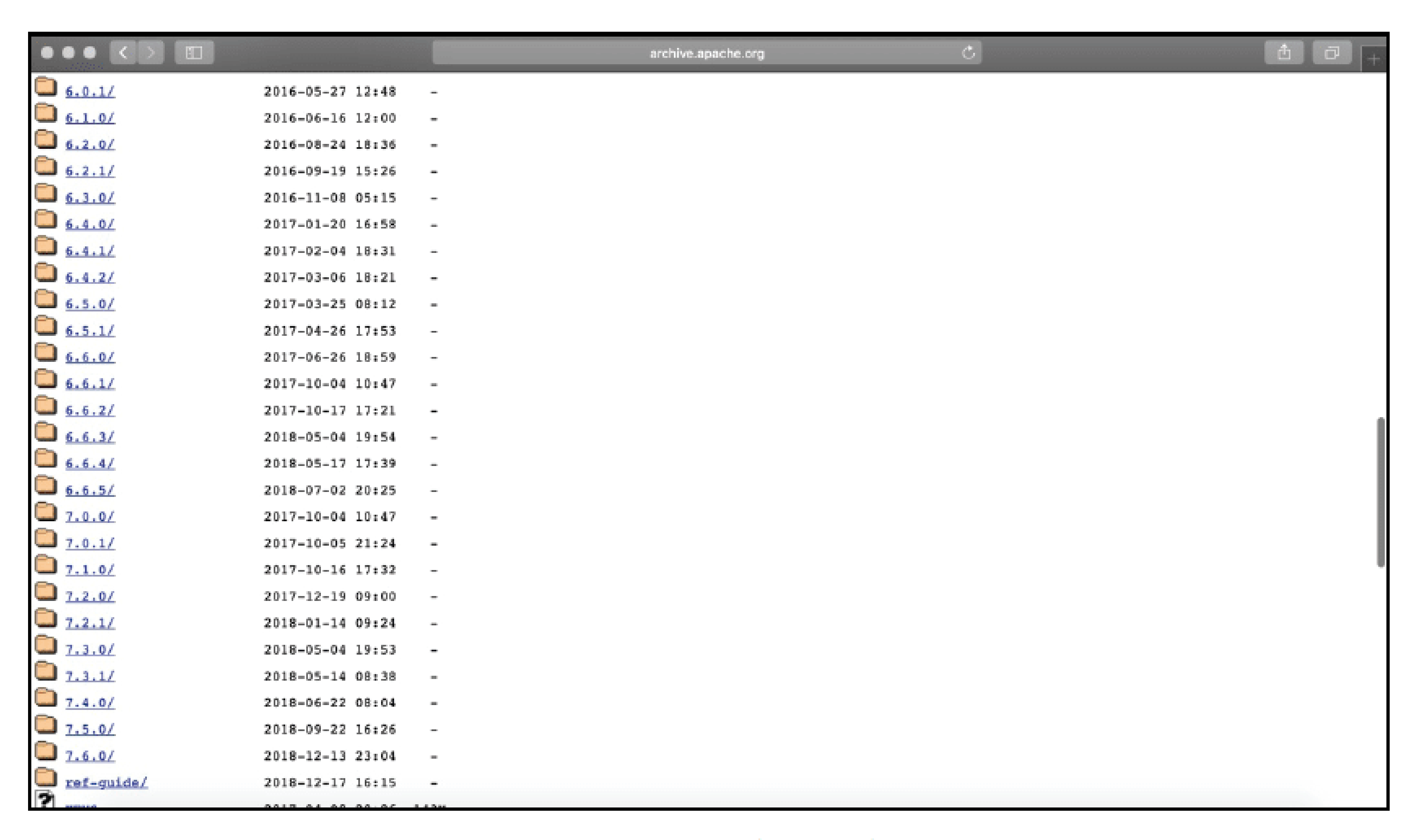


图 10-4　Solr 官网版本

他的首个迭代版本叫作 Compass。第二个迭代版本就是 Elasticsearch（基于 Apache Lucene 开发）。他将 Elasticsearch 作为开源产品发布给公众，并创建了 Elasticsearch IRC 通道，接着就静待用户出现了。

公众反响十分强烈。用户自然而然地喜欢上了这个软件。由于使用量急速攀升，这个软件开始有了自己的社区，并引起了人们的高度关注，尤其引发了 Steven Schuurman、Uri Boness 和 Simon Willnauer 的浓厚兴趣。他们 4 人最终共同组建了一家搜索公司。

在 Elastic 官网（地址：https://www.elastic.co/cn/about/history-of-elasticsearch）上记载着 Elasticsearch 辉煌的发展史。在 2018 年美国时间 10 月 5 号，Elasticsearch 在美国上市，在上市到现在的几个月内，官网（地址：https://www.elastic.co/downloads/past-releases）不断更新版本，如图 10-5 所示。

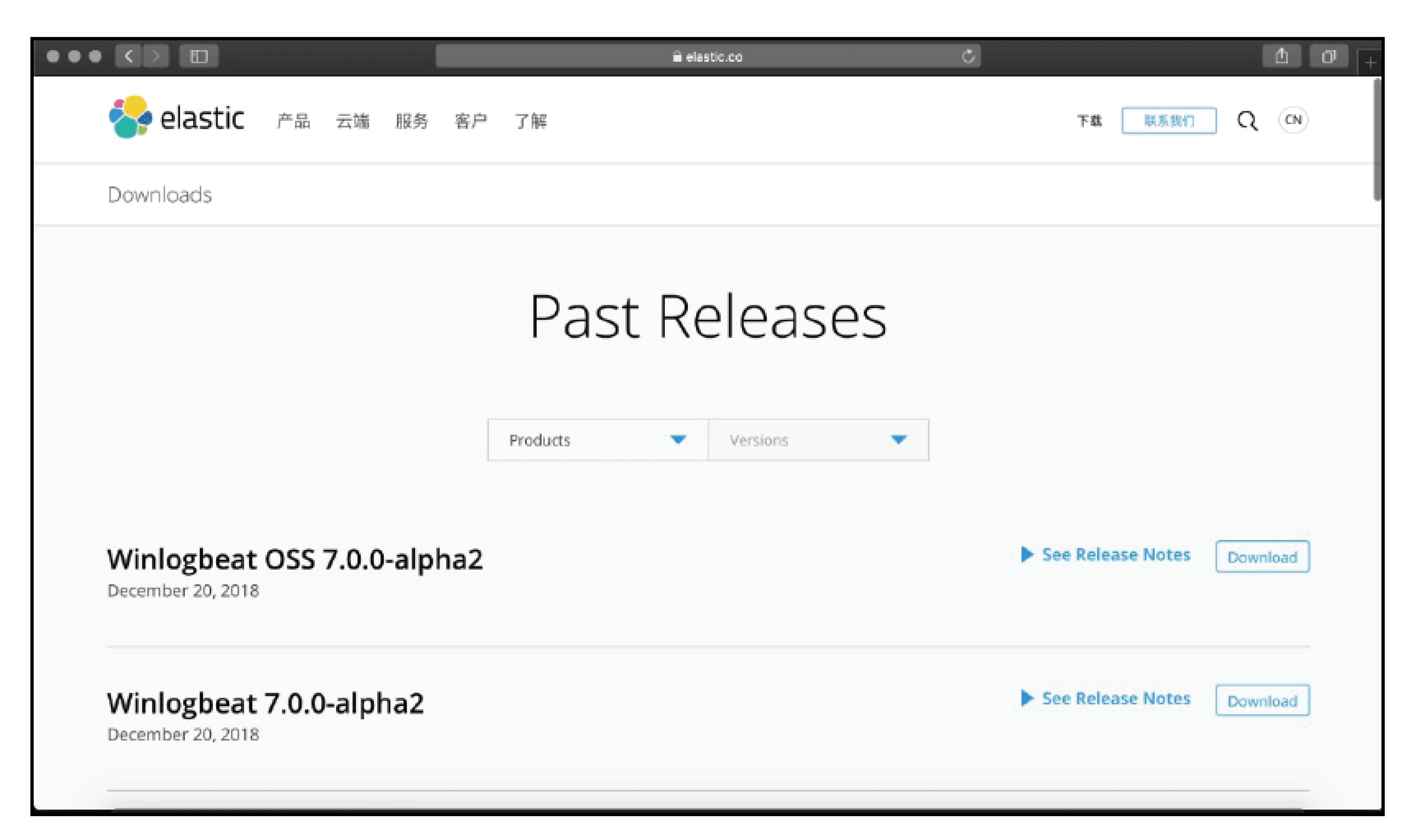


图 10-5　Elasticsearch 官网版本

综上所述，对于二者的技术背景，笔者认为可以打个平手，毕竟两个搜索引擎框架背后都是非常成熟的公司，并且相信在未来对二者的关注会持续保持高涨。

10.3.2 热度比较

在百度搜索指数中，开发人员近些年对 Elasticsearch 的关注程度及搜索指数更高一些，2011 年初到 2018 年底的百度搜索指数如图 10-6 所示。



图 10-6 百度搜索指数

在谷歌搜索指数中，也是 Elasticsearch 在近些年的关注度更胜一筹，2004 年初到 2018 年底，谷歌搜索指数如图 10-7 所示。



图 10-7 谷歌搜索指数

综上所述，对于二者的热度比较，Elasticsearch 更胜一筹。

10.3.3　集群部署

在部署方面，只要下载 Solr 可以运行在 Web 服务器上的可执行文件即可；在集群方面，需要使用 Zookeeper 进行集群管理。

Elasticsearch 单节点安装比较简单，下载对应版本的压缩文件，解压即可使用。在集群方面，Elasticsearch 有一个内置的类似 ZooKeeper 的名为 Zen 的组件，通过内部的协调机制来维护集群状态。在集群的时候需要注意各项配置，稍不注意就可能遇到一些“坑”。

综上所述，对于集群部署方面，由于笔者在部署 Elasticsearch 集群时踩过一些“坑”，因此个人认为 Solr 稍微简单一些。

10.3.4　数据格式

Solr 支持多种数据格式，如 JSON、XML、CSV 等，但是 Elasticsearch 仅支持 JSON 文件格式，在数据格式方面，Solr 完胜。

10.3.5　效率

这个观点是开发者对二者选型最具决定性的一点，当对已经创建好索引的文件进行查询时，Solr 的效率明显高于 Elasticsearch。但是当查询实时数据时，由于 Solr 创建索引时会产生 IO 阻塞，因此查询性能很差，在这种情况下，Elasticsearch 的效率明显高于 Solr。

综上所述，如果业务场景需要实时搜索，那么建议选择 Elasticsearch；如果是对已有数据进行查询，并且改变不大，那么建议使用 Solr。

10.4　小　　结

本章对常用的两种搜索引擎进行了介绍，并且基于 Spring Boot 对二者进行了一定的使用。相信经过本章的学习，读者会对二者的使用有一定的认知，进而在实际项目中应用。

第 11 章

Spring Boot 的小彩蛋

本章学习 Spring Boot 的一些扩展功能，有些功能可能在实际工作中没有用处，但是有些功能在实际工作中使用得非常多，需要我们密切关注。

11.1 修改启动 Banner

Spring Boot 应用程序的启动 Banner 是一大亮点，本节带领大家学习如何修改启动 Banner。

11.1.1 启动 Banner 介绍

在第一次接触 Spring Boot 应用的时候，除了其性能、配置上的亮点外，更吸引笔者注意的就是启动 Banner，如图 11-1 所示。

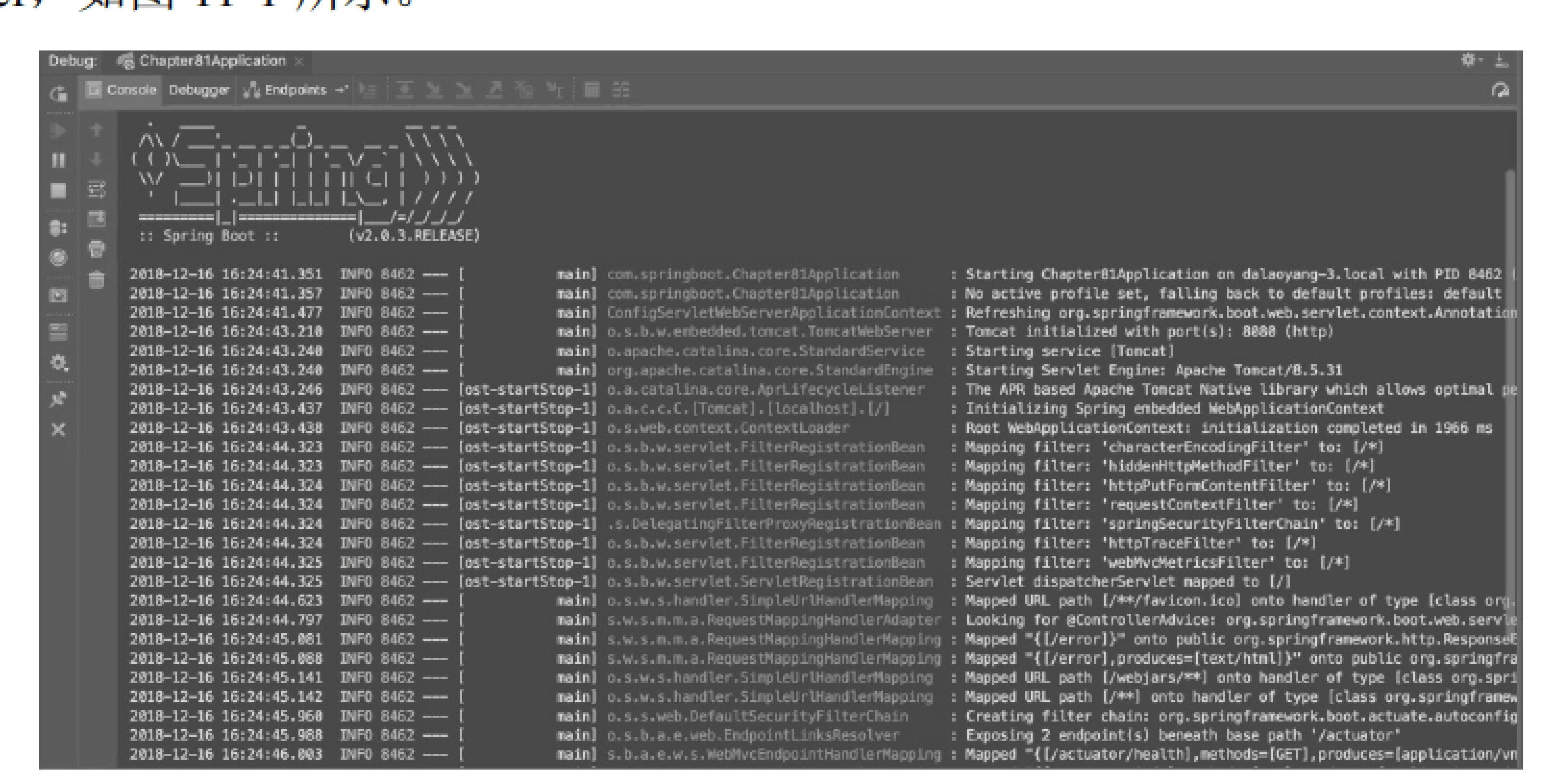


图 11-1 Spring Boot 启动 Banner

Spring Boot 应用程序在初始化的时候使用 SpringApplication 类的 run 方法，如代码清单 11-1 所示。

代码清单 11-1　Spring Boot 应用启动时初始化的方法

```
    public ConfigurableApplicationContext run(String... args) {
            StopWatch stopWatch = new StopWatch();
            stopWatch.start();
            ConfigurableApplicationContext context = null;
            Collection<SpringBootExceptionReporter> exceptionReporters = new
ArrayList<>();
            configureHeadlessProperty();
            SpringApplicationRunListeners listeners = getRunListeners(args);
            listeners.starting();
            try {
                ApplicationArguments applicationArguments = new
DefaultApplicationArguments(args);
                ConfigurableEnvironment environment =
prepareEnvironment(listeners, applicationArguments);
                configureIgnoreBeanInfo(environment);
                Banner printedBanner = printBanner(environment);
                context = createApplicationContext();
                exceptionReporters = getSpringFactoriesInstances(
                        SpringBootExceptionReporter.class,
                        new Class[] { ConfigurableApplicationContext.class },
context);
                prepareContext(context, environment, listeners,
applicationArguments, printedBanner);
                refreshContext(context);
                afterRefresh(context, applicationArguments);
                stopWatch.stop();
                if (this.logStartupInfo) {
                    new StartupInfoLogger(this.mainApplicationClass)
                            .logStarted(getApplicationLog(), stopWatch);
                }
                listeners.started(context);
                callRunners(context, applicationArguments);
            }
            catch (Throwable ex) {
                handleRunFailure(context, ex, exceptionReporters, listeners);
                throw new IllegalStateException(ex);
            }

            try {
                listeners.running(context);
```

```
        }
        catch (Throwable ex) {
            handleRunFailure(context, ex, exceptionReporters, null);
            throw new IllegalStateException(ex);
        }
        return context;
    }
```

从源代码中可以看出，启动 Banner 调用了当前类中的 printBanner 方法。在该方法中，首先在 Classpath 内依次寻找 banner.gif、banner.jpg、banner.png 和 banner.txt 文件，如果找到，就使用；如果没找到；就使用默认的 Spring Boot Banner。其中，printBanner()方法如代码清单 11-2 所示。

代码清单 11-2　Spring Boot 应用启动时调用 printBanner 方法

```
    private Banner printBanner(ConfigurableEnvironment environment) {
        if (this.bannerMode == Banner.Mode.OFF) {
            return null;
        }
        ResourceLoader resourceLoader = (this.resourceLoader != null ?
this.resourceLoader : new DefaultResourceLoader(getClassLoader()));
        SpringApplicationBannerPrinter bannerPrinter = new
SpringApplicationBannerPrinter(resourceLoader, this.banner);
        if (this.bannerMode == Mode.LOG) {
            return bannerPrinter.print(environment, this.mainApplicationClass,
logger);
        }
        return bannerPrinter.print(environment, this.mainApplicationClass,
System.out);
    }
```

刚刚说到，默认会打印 Spring Boot Banner，这个类通过将 Spring 的 Logo 写成一个字符串数组，然后通过 String Builder 打印出来，实现启动 Banner 的打印。其中 SpringBootBanner 类的完整代码如代码清单 11-3 所示。

代码清单 11-3　Spring Boot 应用-Spring Boot Banner 类完整代码

```
    class SpringBootBanner implements Banner {

        private static final String[] BANNER = { "",
                "  .   ____          _            __ _ _",
                " /\\\\ / ___'_ __ _ _(_)_ __  __ _ \\ \\ \\ \\",
                "( ( )\\___ | '_ | '_| | '_ \\/ _` | \\ \\ \\ \\",
                " \\\\/  ___)| |_)| | | | | || (_| |  ) ) ) )",
                "  '  |____| .__|_| |_|_| |_\\__, | / / / /",
                " =========|_|==============|___/=/_/_/_/" };
```

```
    private static final String SPRING_BOOT = " :: Spring Boot :: ";

    private static final int STRAP_LINE_SIZE = 42;

    @Override
    public void printBanner(Environment environment, Class<?> sourceClass,
            PrintStream printStream) {
        for (String line : BANNER) {
            printStream.println(line);
        }
        String version = SpringBootVersion.getVersion();
        version = (version != null ? " (v" + version + ")" : "");
        StringBuilder padding = new StringBuilder();
        while (padding.length() < STRAP_LINE_SIZE
                - (version.length() + SPRING_BOOT.length())) {
            padding.append(" ");
        }

        printStream.println(AnsiOutput.toString(AnsiColor.GREEN,
                SPRING_BOOT, AnsiColor.DEFAULT, padding.toString(),
                AnsiStyle.FAINT, version));
        printStream.println();
    }

}
```

11.1.2　启动 Banner 修改

前面已经介绍了，修改启动 Banner 就是在 scr/main/resources 文件夹下创建一个对应的 Banner 文件。这里创建一个 banner.txt 文件，启动后如图 11-2 所示。



图 11-2　修改后的 Spring Boot 应用程序启动 Banner

另外，Spring Boot 还提供了几个配置来设置 Banner，分别说明如下。

- ${AnsiColor.BRIGHT_CYAN}：设置 Banner 字体颜色。
- ${AnsiBackground.BRIGHT_CYAN}:设置 Banner 背景颜色。
- ${AnsiStyle.UNDERLINE}：设置字体样式。
- ${application.version}：对应显示配置文件中 application.version 的属性配置，主要用于配置版本号。
- ${application.formatted-version}：用于格式化版本号，默认在版本号后面加 v。
- ${spring-boot.version}：Spring Boot 的版本号。

- ${spring-boot.formatted-version}：用于格式化 Spring Boot 版本号，默认格式如(v2.0.3.RELEASE)。

当然，可能有人不喜欢这个 Banner，Spring Boot 提供了关闭 Banner 的开关。在启动类设置 setBannerMode(Banner.Mode.OFF)，即可关闭开关，如代码清单 11-4 所示。

代码清单 11-4　Spring Boot 应用-启动类关闭 Banner 代码

```
@SpringBootApplication
public class Chapter101Application {
    public static void main(String[] args) {
        SpringApplication springApplication = new SpringApplication
(Chapter101Application.class);
        springApplication.setBannerMode(Banner.Mode.OFF);
        springApplication.run(args);
    }
}
```

在 Spring Boot 2.0 以后，支持使用动态 GIF 当作 Banner，比如在 src/mian/resources 下放入 banner.gif，启动项目时会优先播放这个 GIF，如果同时存在 banner.gif 和 banner.txt，则会优先播放 GIF，再打印 TXT 文件中的内容。感兴趣的读者可以自己创建漂亮的启动 Banner，动手试试吧！

11.2　使用 LomBok 让编程更简单

开发过程中会创建非常多的实体类，反复使用 IDE 对实体类的 Set 方法、Get 方法进行创建十分麻烦。本节带领大家学习使用 Lombok 让编程变得简单。

11.2.1　什么是 LomBok

Project LomBok（官网地址：https://www.projectlombok.org/）是一个简化编程的 Java 库，通过使用它可以利用注解的形式省去写 Set 方法、Get 方法、构造函数、equals 方法、toString 方法。举个例子，使用 User 类时，如果没有使用 LomBok，每新增一个属性，就需要反复写 Set 方法、Get 方法，修改构造函数，等等。但是使用 LomBok 后，一个@getter 就可以给所有属性添加 Get 方法，甚至使用@Data 方法可以为 JavaBean 自动注入所有属性的 Set 方法、Get 方法以及构造函数。同时，LomBok 还提供了对日志打印的方法，从而可以大大地简化冗余的 JavaBean 代码。

11.2.2　IntelliJ IDEA 安装 Lombok 插件

在 IntelliJ IDEA 中安装 Lombok 插件有两种方式，第一种是在 IntelliJ IDEA 中直接安装插件，如图 11-3 所示。

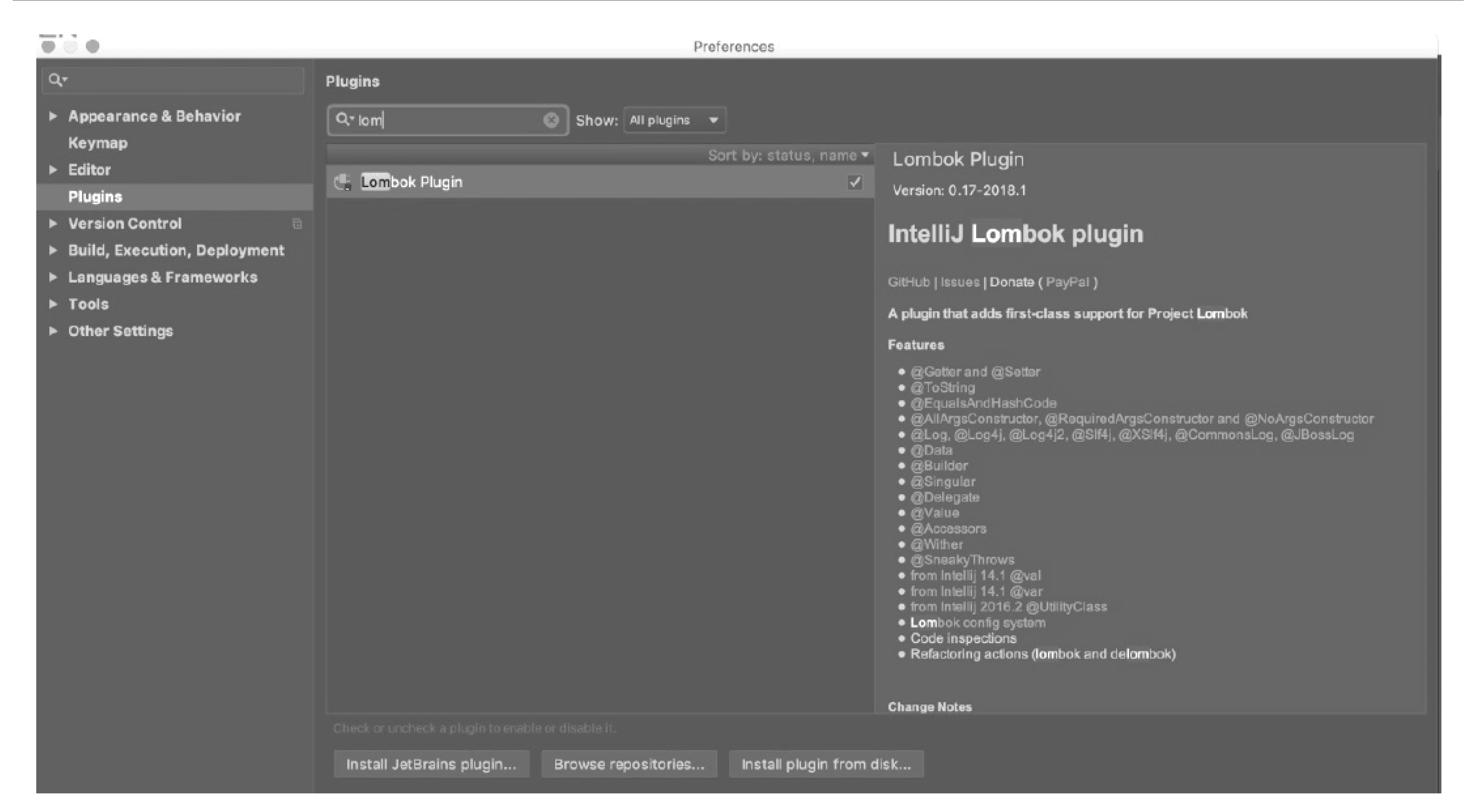


图 11-3　IntelliJ IDEA 内安装 Lombok 插件

第二种是通过 IntelliJ IDEA 官网插件页（http://plugins.jetbrains.com/）下载，然后放入 IntelliJ IDEA 安装目录的 plugins 文件夹内。两种方式无论选择哪种，安装后都需要重启 IntelliJ IDEA。

11.2.3　如何使用 LomBok

在 Spring Boot 应用程序中使用 Lombok 非常简单。新建项目，在项目中引入 lombok 依赖文件，如代码清单 11-5 所示。

代码清单 11-5　Spring Boot-Lombok 项目-引入 lombok 依赖代码

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.20</version>
 </dependency>
```

接下来在对应实体类中加入注解即可实现诸如 Set 方法、Get 方法的自动生成，代码这里不作展示。下面对 Lombok 注解进行介绍。

- val：用在局部变量前面，相当于将变量声明为 final。
- @NonNull：为方法参数增加这个注解会自动对参数进行是否为空的校验，如果为空，就会抛出 NullPointerException 异常。
- @CleanUp：自动管理资源，用在局部变量之前，在当前变量范围内，即将执行完毕退出之前会自动清理资源，自动生成 try-finally 这样的代码来关闭流。
- @Setter/@Getter：用在属性上之后自动为其生成 Set 和 Get 方法。

- @ToString：在类上使用，使用后自动生成 toString 方法。默认情况下，它会按顺序打印类的名称以及每个字段，并以逗号分隔。并且提供了 exclude 属性和 callSuper 属性，其中可以使用 exclude 属性排除属性，比如@ToString(exclude="name")可以排除 name 属性；callSuper 属性设置为 True 后可以将父类的所有 toString 输出。
- @EqualsAndHashcode：用在类上，自动从对象的字段中生成 hashCode 和 equals 的实现 。
- @NoArgsConstructor：用在类上，自动生成一个没有参数的构造函数方法。
- @RequiredArgsConstructor：用在类上，自动生成一个包含常量（final）和标识了@NotNull 的变量的构造方法。
- @AllArgsConstructor：用在类上，为所有字段生成带有一个参数的构造方法。
- @Data：用在类上，其实@Data 是一个组合注解，其包含@ToString、@EqualsAndHashCode、@Getter、@Setter、@RequiredArgsConstructor 注解的特征。简单来说，@Data 注解可以满足一个 JavaBean 的基本需求，对于 POJO 类十分有用。
- @Value：用在类上，是@Data 的不可变形式，相当于为属性添加 final 声明，只提供 getter 方法，而不提供 setter 方法。
- @Builder：用在类、构造器、方法上，提供复杂的 builder APIs。
- @SneakyThrows：用在方法上，自动为方法加入 try、catch 检查异常。
- @Synchronized：用在方法上，将方法声明为同步的，并自动加锁，而锁对象是一个私有的属性$lock 或$LOCK。Java 中 Syncronized 关键字无论是加在方法上还是对象上，所获取的锁都是对象。
- @Log：支持各种 logger 对象，使用时用对应的注解，如@Log4j、@Log4j2、@CommonsLog、@Log、@Slf4j、@XSlf4j。

11.3 邮件发送

对于开发者来说，邮件发送是一个老生常谈的问题了。比如系统出现异常，定期向维护人员发送报告、定期的总结等，需要使用邮件发送的时候非常多。

在 Spring Boot 框架中，为我们提供了便捷使用的 starter 来实现邮件发送功能。本节将学习在 Spring Boot 中使用邮件发送。

11.3.1 在 Spring Boot 中使用邮件发送

在 Spring Boot 应用中，想要使用邮件发送功能，就要在 pom 文件中加入 spring-boot-starter-mail 依赖，如代码清单 11-6 所示。

代码清单 11-6 Spring Boot-Mail 项目依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

11.3.2 基础配置信息

接下来需要在配置文件中配置邮箱信息，如发送邮箱服务器地址、邮箱用户名、邮箱密码，如果使用的邮箱含有独立密码，就需要在密码处填写独立密码。这里以阿里云邮箱为例，配置如代码清单 11-7 所示。

代码清单 11-7 Spring Boot-Mail 项目配置文件代码

```
##邮箱服务器地址
##QQ smtp.qq.com
##sina smtp.sina.cn
##aliyun smtp.aliyun.com
##163 smtp.163.com
spring.mail.host=smtp.aliyun.com
##邮箱用户名
spring.mail.username=dalaoyang@aliyun.com
##邮箱密码（注意：qq 邮箱应该使用独立密码，去 qq 邮箱设置中获取）
spring.mail.password=******
##编码格式
spring.mail.default-encoding=UTF-8

##发送邮件地址
mail.fromMail.sender=dalaoyang@aliyun.com
##接收邮件地址
mail.fromMail.receiver=yangyang@dalaoyang.cn
```

这里以 Controller 调用为例，Spring Boot 使用邮件发送都是操作 JavaMailSender 类来实现的，所以我们创建一个 MailController 类，在类中注入发送邮件关键的类 JavaMailSender，并且使用 sender 和 receiver 分别接收配置文件配置的发送者和接收者，如代码清单 11-8 所示。

代码清单 11-8 Spring Boot-Mail 项目测试类基础代码

```
@RestController
public class MailController {

    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    @Value("${mail.fromMail.sender}")
    private String sender;

    @Value("${mail.fromMail.receiver}")
    private String receiver;
```

```
        //内容接下来详细介绍
        ...
```

11.3.3 文本邮件发送

邮件发送中最简单的就是发送文本邮件。使用 JavaMailSender 发送文本邮件很简单，创建一个 SimpleMailMessage 实体，在实体类中设置以下几个参数即可。

- from：邮件发送地址。
- to：邮件接收者，可以是多个，使用逗号分隔。
- subject：邮件的主题。
- text：邮件的正文内容。

了解如何使用后，接下来在 MailController 中创建一个 simpleMail 方法测试文本邮件发送，如代码清单 11-9 所示。

代码清单 11-9　Spring Boot-Mail 项目发送普通文本代码

```
@GetMapping("simpleMail")
public void simpleMail(){
    String subject = "文本邮件";
    String text = "文本邮件正文内容";
    SimpleMailMessage simpleMailMessage = new SimpleMailMessage();
    simpleMailMessage.setFrom(sender);
    simpleMailMessage.setTo(receiver);
    simpleMailMessage.setSubject(subject);
    simpleMailMessage.setText(text);
    try {
        javaMailSender.send(simpleMailMessage);
        logger.info("发送文本邮件成功！");
    } catch (Exception e) {
        logger.error("发送文本邮件时发生异常！", e);
    }
}
```

启动项目，发送 HTTP 请求 http://localhost:8080/simpleMail，稍等一下查看邮件，可以看到如图 11-4 所示的内容。

图 11-4　发送文本邮件

收到这个邮件后，文本邮件就发送完成了。

11.3.4 网页邮件发送

网页邮件发送是指发送一段符合 HTML 语法的字符串。使用网页邮件发送的时候需要操作 MimeMessage 实体，在实体类中设置以下几个参数。

- from：邮件发送地址。
- to：邮件接收者，可以是多个，使用逗号分隔。
- subject：邮件的主题。
- text：第一个参数拼接好的 HTML 字符串，第二个参数为 true。

在 MailController 中创建 htmlMail 方法来测试发送网页邮件，如代码清单 11-10 所示。

代码清单 11-10 Spring Boot-Mail 项目发送网页邮件代码

```
@GetMapping("/htmlMail")
public void htmlMail() {
    String subject = "网页邮件";
    String content="<html>\n" +
        "<body>\n" +
        "    <h3>这是一封 Html 邮件!</h3>\n" +
        "</body>\n" +
        "</html>";
    MimeMessage message = javaMailSender.createMimeMessage();
    try {
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setFrom(sender);
        helper.setTo(receiver);
        helper.setSubject(subject);
        helper.setText(content, true);
        javaMailSender.send(message);
        logger.info("发送网页邮件成功！");
    } catch (Exception e) {
        logger.error("发送网页邮件时发生异常！", e);
    }
}
```

启动项目，发送 HTTP 请求 localhost:8080/htmlMail，查看邮件，可以看到如图 11-5 所示的内容。

网页邮件 ☆
发件人：dalaoyang <dalaoyang@aliyun.com>
时　间：2019年1月13日(星期天) 中午12:44
收件人：大老杨。<yangyang@dalaoyang.cn>
纯文本

这是一封Html邮件!

图 11-5　发送网页邮件

收到这个邮件后，网页邮件就发送完成了。

11.3.5　附件邮件发送

发送邮件的时候经常会遇到附件。接下来，我们学习如何发送附件邮件，与发送网页邮件使用的实体一样，只不过利用 addAttachment 方法将文件添加进来。举个例子，笔者要将电脑桌面上的 settings.xml 文件发送到邮件中，发送多个邮件就多次调用 addAttachment 方法，如代码清单 11-11 所示。

代码清单 11-11　Spring Boot-Mail 项目发送附件邮件代码

```
@RequestMapping("/filesMail")
public void filesMail() {
    String subject = "附件邮件";
    String text = "附件邮件正文内容";
    String filePath="/Users/dalaoyang/Desktop/settings.xml";
    MimeMessage message = javaMailSender.createMimeMessage();
    try {
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setFrom(sender);
        helper.setTo(receiver);
        helper.setSubject(subject);
        helper.setText(text, true);
        FileSystemResource file = new FileSystemResource(new File(filePath));
        String fileName =
filePath.substring(filePath.lastIndexOf(File.separator));
        helper.addAttachment(fileName, file);
        javaMailSender.send(message);
        logger.info("发送附件邮件成功！");
    } catch (Exception e) {
        logger.error("发送附件邮件时发生异常！", e);
    }
}
```

启动项目，发送 HTTP 请求 localhost:8080/filesMail，查看邮件，可以看到如图 11-6 所示的内容。



图 11-6　发送附件邮件

收到这个邮件后，附件邮件就发送完成了。

11.3.6　嵌入静态资源邮件发送

嵌入静态资源邮件其实是网页邮件的改进版本，通俗来说，就在网页邮件的图片标签中嵌入图片。举个例子，笔者要将电脑桌面上的 mail.jpg 图片嵌入邮件中，若需要嵌入多个图片，则直接在页面拼接多个图片标签即可，如代码清单 11-12 所示。

代码清单 11-12　Spring Boot-Mail 项目发送嵌入静态资源邮件代码

```
@RequestMapping("/inlineResourceMail")
public void inlineResourceMail() {
    String Id = "test001";
    String subject = "嵌入静态资源邮件";
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("<html><body>这是有图片的邮件：");
    stringBuilder.append("<img src='cid:" + Id + "' >");
    stringBuilder.append("</body></html>");
    String content = stringBuilder.toString();
    String imgPath = "/Users/dalaoyang/Desktop/mail.jpg";
    MimeMessage message = javaMailSender.createMimeMessage();
    try {
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setFrom(sender);
        helper.setTo(receiver);
        helper.setSubject(subject);
        helper.setText(content, true);
        FileSystemResource res = new FileSystemResource(new File(imgPath));
        helper.addInline(Id, res);
        javaMailSender.send(message);
        logger.info("嵌入静态资源邮件成功！");
```

```
        } catch (Exception e) {
            logger.error("发送嵌入静态资源邮件时发生异常！", e);
        }
    }
```

启动项目，发送 HTTP 请求 localhost:8080/inlineResourceMail，查看邮件，可以看到如图 11-7 所示的内容。

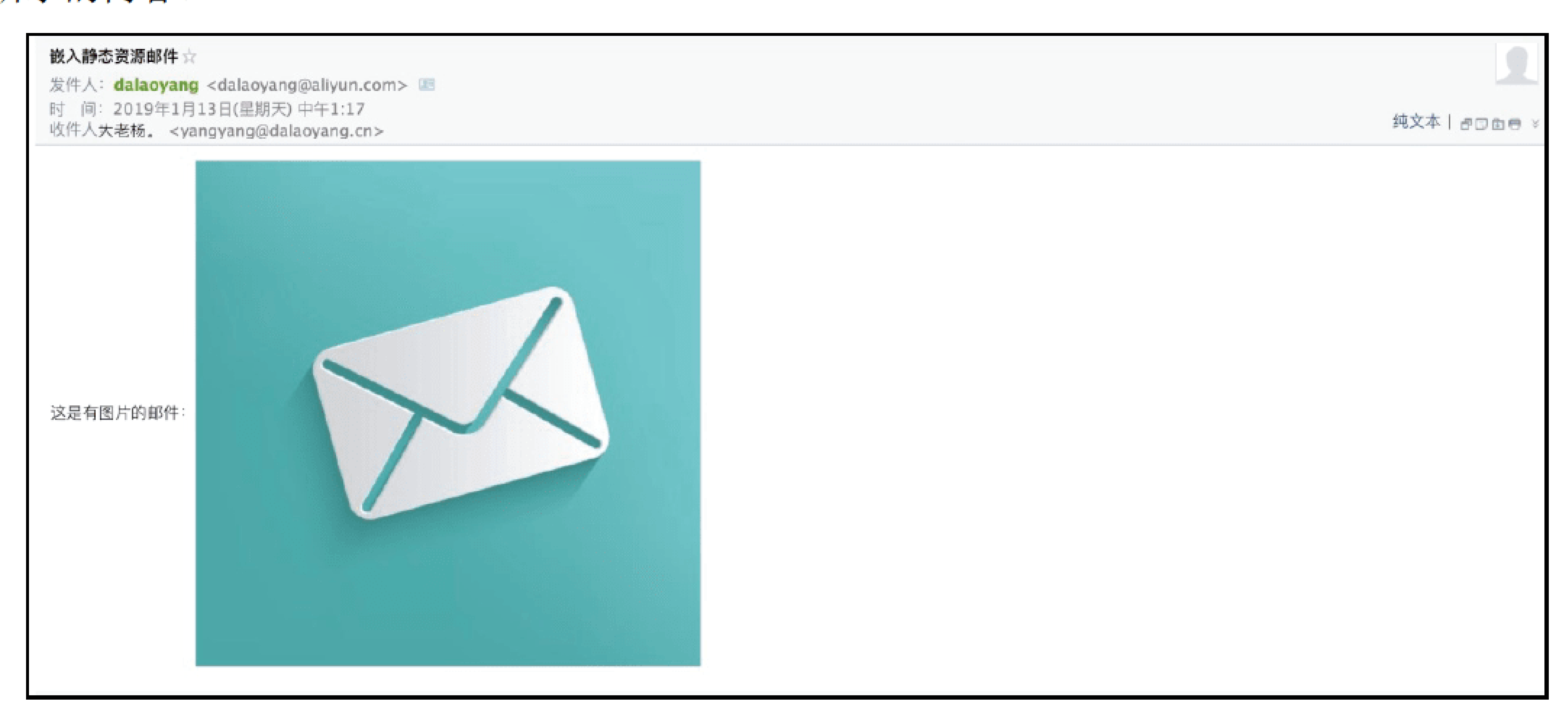


图 11-7　发送嵌入静态资源邮件

收到这个邮件后，嵌入静态资源邮件就发送完成了。邮件发送大致就这几种类型，读者可以根据实际情况结合使用。

11.4　三“器”的使用

在实际应用开发中，可能存在一些与业务无关，但是起着重要作用的功能，比如在程序中校验用户是否登录、是否有权限去做这件事、进行统一的日志打印、异常处理等。这时就可以利用强大的三“器”来实现这些功能。

可能很多人已经猜到了，笔者所指的三“器”就是耳熟能详的过滤器、拦截器和监听器。本节带领大家学习在 Spring Boot 中使用过滤器、拦截器和监听器。

11.4.1　过滤器

本小节学习三“器”的第一个主解 —— 过滤器。

1. 过滤器介绍

过滤器的英文名称为 Filter，是 Servlet 技术中最实用的技术。如同它的名字一样，过滤器是处于客户端与服务器资源文件之间的一道过滤网，帮助我们过滤一些不符合要求的请求。通常它被用

作 Session 校验，判断用户权限，如果不符合设定条件，就会被拦截到特殊的地址或者给予特殊的响应。

2. 使用过滤器

使用过滤器很简单，只需要实现 Filter 类，然后重写它的 3 个方法即可。

- init 方法：在容器中创建当前过滤器的时候自动调用这个方法。
- destory 方法：在容器中销毁当前过滤器的时候自动调用这个方法。
- doFilter 方法：这个方法有 3 个参数，分别是 ServletRequest、ServletResponse 和 FilterChain。可以从参数中获取 HttpServletRequest 和 HttpServletResponse 对象进行相应的处理操作。

接下来，我们直接创建一个过滤器 MyFilter。这里做一些 URL 的拦截，如果符合条件，就正常跳转，如果不符合条件，就拦截到/online 请求中。MyFilter 完整内容如代码清单 11-13 所示。

代码清单 11-13　Spring Boot-过滤器项目代码

```
public class MyFilter implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
        System.out.println("MyFilter 被调用");
        HttpServletRequest httpServletRequest = (HttpServletRequest) request;
        HttpServletResponseWrapper wrapper = new HttpServletResponseWrapper
((HttpServletResponse) response);
        //只有符合条件的可以直接请求，不符合的跳转到/online 请求中
        String requestUri = httpServletRequest.getRequestURI();
        System.out.println("请求地址是："+requestUri);
        if (requestUri.indexOf("/addSession") != -1
                || requestUri.indexOf("/removeSession") != -1
                || requestUri.indexOf("/online") != -1
                || requestUri.indexOf("/favicon.ico") != -1) {
            chain.doFilter(request, response);
        } else {
            wrapper.sendRedirect("/online");
        }
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        //在服务启动时初始化
        System.out.println("初始化拦截器");
    }
```

```
    @Override
    public void destroy() {
        //在服务关闭时销毁
        System.out.println("销毁拦截器");
    }
}
```

11.4.2 拦截器

本小节学习三“器”的第二个主角 —— 拦截器。

1. 拦截器介绍

Java 中的拦截器是动态拦截 action 调用的对象，然后提供了可以在 action 执行前后增加一些操作，也可以在 action 执行前停止操作。其实拦截器也可以做和过滤器同样的操作，以下是拦截器的常用场景。

- 登录认证：在一些简单应用中，可能会通过拦截器来验证用户的登录状态，如果没有登录或者登录失效，就会给用户一个友好的提示或者返回登录页面。
- 记录系统日志：在 Web 应用中，通常需要记录用户的请求信息，比如请求的 IP、方法执行时常等，通过这些记录可以监控系统的状况，以便于对系统进行信息监控、信息统计、计算 PV（Page View）和性能调优等。
- 通用处理：在应用程序中可能存在所有方法都要返回的信息，这时可以使用拦截器来实现，省去每个方法冗余重复的代码实现。

2. 使用拦截器

这里以使用 Spring 拦截器为例，在类上需要实现 HandlerInterceptor 类，并且重写类中的 3 个方法，分别是：

- preHandle 在业务处理器处理请求之前被调用，返回值是 boolean 值，如果返回 true，就进行下一步操作；若返回 false，则证明不符合拦截条件。在失败的时候不会包含任何响应，此时需要调用对应的 response 返回对应响应。
- postHandle 在业务处理器处理请求执行完成后、生成视图前执行。这个方法可以通过方法参数 ModelAndView 对视图进行处理，当然 ModelAndView 也可以设置为 null。
- afterCompletion 在 DispatcherServlet 完全处理请求后被调用，通常用于记录消耗时间，也可以进行一些资源处理操作。

接下来，创建一个自定义拦截器 MyInterceptor，我们用这个拦截器打印请求的耗时，并且判断当前的浏览器是否存在 Session。MyInterceptor 完整内容如代码清单 11-14 所示。

代码清单 11-14 Spring Boot-拦截器项目代码

```
    @Component
    public class MyInterceptor implements HandlerInterceptor {
        @Override
        public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
            System.out.println("preHandle 被调用");
            request.setAttribute("startTime", System.currentTimeMillis());
            return true;
        }

        @Override
        public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
            System.out.println("postHandle 被调用");
            HttpSession session = request.getSession();
            String name = (String) session.getAttribute("name");
            if("dalaoyang".equals(name)){
                System.out.println("--------------------当前浏览器存在 session");
            }
        }

        @Override
        public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws Exception {
            System.out.println("afterCompletion 被调用");
            long startTime = (Long) request.getAttribute("startTime");
            System.out.println("--------------------请求耗时：" +
(System.currentTimeMillis() - startTime));
        }
    }
```

11.4.3 监听器

本小节学习三“器”的最后一个主角——监听器。

1. 监听器介绍

监听器通常用于监听 Web 应用中对象的创建、销毁等动作的发生，同时对监听的情况做出相应的处理，最常用于统计网站的在线人数、访问量等信息。

监听器大致分为以下几种。

- ServletContextListener：用来监听 ServletContext 属性的操作，比如新增、修改、删除。

- HttpSessionListener：用来监听 Web 应用中的 Session 对象，通常用于统计在线情况。
- ServletRequestListener：用来监听 Request 对象的属性操作。

2. 使用监听器

使用监听器的话，只需要在类中实现对应功能的监听器对象，如本文使用的 HttpSessionListener。下面以监听 Session 信息为例统计在线人数。新建一个 MyHttpSessionListener 类，实现 HttpSessionListener 类，在类中定义一个全局变量 online，当创建 Session 时，online 的数量加 1；当销毁 Session 时，online 的数量减 1。MyHttpSessionListener 完整内容如代码清单 11-15 所示。

代码清单 11-15 Spring Boot-监听器项目代码

```
public class MyHttpSessionListener implements HttpSessionListener {

    public static int online = 0;

    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.out.println("sessionCreated 被调用");
        online ++;
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        System.out.println("sessionDestroyed 被调用");
        online --;
    }
}
```

11.4.4 Spring Boot 引用三“器”

过滤器、监听器和拦截器我们已经创建好了，但是还不能引用。接下来我们修改 Spring Boot 启动类，在类中通过注入 Bean 的方式引用三者。启动类完整内容如代码清单 11-16 所示（也可以使用其他方法引用，比如创建一个配置类统一管理）。

代码清单 11-16 Spring Boot-引用三器代码

```
@SpringBootApplication
public class Chapter114Application implements WebMvcConfigurer {

    public static void main(String[] args) {
        SpringApplication.run(Chapter114Application.class, args);
    }
```

```
    @Autowired
    private MyInterceptor myInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(myInterceptor);
    }

    @Bean
    public FilterRegistrationBean filterRegist() {
        FilterRegistrationBean frBean = new FilterRegistrationBean();
        frBean.setFilter(new MyFilter());
        frBean.addUrlPatterns("/*");
        return frBean;
    }

    @Bean
    public ServletListenerRegistrationBean listenerRegist() {
        ServletListenerRegistrationBean srb = new
ServletListenerRegistrationBean();
        srb.setListener(new MyHttpSessionListener());
        return srb;
    }
}
```

11.4.5　测试

接下来，使用 Controller 对过滤器、拦截器和监听器进行测试，在类中我们只需要创建 3 个方法，分别说明如下。

- addSession：负责新增一个 Session。
- removeSession：负责销毁当前浏览器的 Session。
- online：查看在线人数。

完整内容如代码清单 11-17 所示。

代码清单 11-17　Spring Boot-三“器”项目测试代码

```
@RestController
public class TestController {

    @GetMapping("/addSession")
    public String addSession(HttpServletRequest request) {
        HttpSession session = request.getSession();
```

```
        session.setAttribute("name", "dalaoyang");
        return "当前在线人数" + MyHttpSessionListener.online;
    }

    @GetMapping("/removeSession")
    public String removeSession(HttpServletRequest request) {
        HttpSession session = request.getSession();
        session.invalidate();
        return "当前在线人数" + MyHttpSessionListener.online;
    }

    @GetMapping("/online")
    public String online() {
        return "当前在线人数：" + MyHttpSessionListener.online + "人";
    }
}
```

启动项目，测试步骤如下：

01 使用浏览器访问 http://localhost:8080/online，可以看到浏览器提示“当前在线人数 0 人”（拦截器成功）。

02 多次使用浏览器访问 http://localhost:8080/addSession，可以看到浏览器提示“当前在线人数 1 人”（监听器成功）。

03 更换浏览器，访问 http://localhost:8080/addSession，可以看到浏览器提示“当前在线人数 2 人”。

04 多次使用浏览器访问 http://localhost:8080/removeSession，可以看到浏览器提示“当前在线人数 1 人”。

05 使用浏览器访问不存在的地址，会自动跳转到 http://localhost:8080/online（过滤器成功）。

从上面的测试可以看出，代码清单中的类限制一个浏览器只能存在一个 Session，在此基础上进行操作。过滤器、拦截器、监听器还可以做很多操作，可以结合实际应用来使用，也可以多个配合使用，不过需要注意设置对应的优先级，以免踩到不必要的“坑”。

11.5 事务使用

11.5.1 事务介绍

事务（Transaction）是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中做的所有更改都会被撤销。也就是事务具有原子性，一个事务中的一系列操作要么全部成功，要么一个都不做。事务的结束有两种，当事务中的所有步骤全部成功执行时，事务提交。如果其中一个步骤失败，就会发生回滚操作，撤销之前做的所有操作，到事务开始。

事务应该具有 4 个属性：原子性、一致性、隔离性、持久性。这 4 个属性通常称为 ACID 特性。

（1）原子性（Atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸多操作要么都做，要么都不做。

（2）一致性（Consistency）。事务必须使数据库从一个一致性状态变成另一个一致性状态。一致性与原子性是密切相关的。

（3）隔离性（Isolation）。一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

（4）持久性（Durability）。持久性也称永久性（Permanence），是指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的，接下来的其他操作或故障不应该对其有任何影响。

11.5.2　在项目中使用事务

下面使用 Spring 声明式事务举一个简单的例子。首先创建一个实体类 Book，注意 bookName 的字段长度为 10，稍后会用到。Book 类内容如代码清单 11-18 所示。

代码清单 11-18　Spring Boot-事务项目实体类代码

```
@Entity
public class Book{

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false,unique = true,length = 10)
    private String bookName;

    ... //省略 set，Get 方法
}
```

接下来以批量插入数据为例，如果不加入事务，异常终止方法就不会回滚，这样会造成一些脏数据的产生。启动项目，使用 JPA 生成表后，bookName 字段的长度是 10，如果插入的 bookName 是“Spring Boot 2 实战之旅”，由于长度的原因，会产生异常并且插入失败，但是由于没有事务的原因，失败前的数据还是会继续插入（如方法 test1），当方法上加入事务注解后，会统一进行数据回滚（如方法 test2），完整内容如代码清单 11-19 所示。

代码清单 11-19　Spring Boot-事务项目测试类代码

```
@RestController
public class BookController {
    @Autowired
    private BookRepository bookRepository;

    @GetMapping("/test1")
```

```
    public String test1(){
        bookRepository.save(new Book("JAVA 从入门到精通"));
        bookRepository.save(new Book("SpringBoot2 实战之旅"));
        return "success";
    }

    @GetMapping("/test2")
    @Transactional
    public String test2(){
        bookRepository.save(new Book("JAVA 从入门到精通"));
        bookRepository.save(new Book("Spring Boot2 实战之旅"));
        return "success";
    }
}
```

启动项目，分别请求两个方法，test1 方法由于没有事务的原因，第一条数据会插入成功，第二条数据会插入失败，这个插入的第一条数据就是我们所说的脏数据；test2 方法由于加入事务的原因，两条数据都不会插入成功，很显然，test2 方法的场景更符合我们事务的特性。

接下来，我们进一步看一下 Spring 事务的特性。

11.5.3 Spring 事务拓展介绍

Spring 事务不仅可以通过使用事务注解@Transactional，同时支持编程式使用事务，但是这种模式不常用。本小节将详细介绍 Spring 事务。

1. 事务隔离级别

使用事务其实只用到了一个注解@Transactional，这就是 Spring 的注解式事务。事务隔离级别是指若干个事务并发时的隔离程度，Spring 声明事务可以通过 isolation 属性来设置 Spring 的事务隔离级别。其中提供了以下 5 种事务隔离级别。

- @Transactional(isolation = Isolation.DEFAULT)：默认的事务隔离级别，即使用数据库的事务隔离级别。
- @Transactional(isolation = Isolation.READ_UNCOMMITTED)：读未提交，这是最低的事务隔离级别，允许其他事务读取未提交的数据，这种级别的事务隔离会产生脏读，不可重复读和幻读。
- @Transactional(isolation = Isolation.READ_COMMITTED)：读已提交，这种级别的事务隔离能读取其他事务已经修改的数据，不能读取未提交的数据，会产生不可重复读和幻读。
- @Transactional(isolation = Isolation.REPEATABLE_READ)：可重复读，这种级别的事务隔离可以防止不可重复读和脏读，但是会发生幻读。
- @Transactional(isolation = Isolation.SERIALIZABLE)：串行化，这是最高级别的事务隔离，会避免脏读，不可重复读和幻读。在这种隔离级别下，事务会按顺序进行。

2. 事务传播行为

事务传播行为是指如果多个事务同时存在，Spring 就会处理这些事务的行为。事务传播行为分为如下几种。

- PROPAGATION_REQUIRED：如果当前存在事务，就加入该事务；如果当前没有事务，就创建一个新的事务，这是 Spring 默认的事务传播行为。
- PROPAGATION_REQUIRES_NEW：创建一个新的事务，如果当前存在事务，就把当前事务挂起。新建事务和被挂起的事务没有任何关系，是两个独立的事务。外层事务回滚失败时，不能回滚内层事务执行结果，内外层事务不能相互干扰。
- PROPAGATION_SUPPORTS：如果当前存在事务，就加入该事务；如果当前没有事务，就以非事务的方式继续运行。
- PROPAGATION_NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，就把当前事务挂起。
- PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，就抛出异常。
- PROPAGATION_MANDATORY：如果当前存在事务，就加入该事务；如果当前没有事务，就抛出异常。
- PROPAGATION_NESTED：如果当前存在事务，就创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，该取值就等价于 PROPAGATION_REQUIRED。

3. 声明式事务属性

Spring 事务不只拥有事务隔离级别和事务传播行为，另外还包含很多属性供开发者使用，分别说明如下。

- value：存放 String 类型的值，主要用来指定不同的事务管理器，满足在同一个系统中存在不同的事务管理器。比如在 Spring 容器中声明了多种事务管理器，然后开发者可以根据设置指定需要使用的事务管理器。通常一个系统需要访问多个数据库的场景下，就会设置多个事务管理器，然后进行不同的选择。
- transactionManager：与 value 类似，也是用来选择事务管理器。
- propagation：事务传播行为，默认值是 Propagation.REQUIRED。
- isolation：事务的隔离级别，默认值是 Isolation.DEFAULT。
- timeout：事务的超时时间，默认值是-1，如果超过了设置的时间还没有执行完成，就会自动回滚当前事务。
- readOnly：当前事务是不是只读事务，默认值是 false。通常可以设置读取数据的事务的属性值为 true。
- rollbackFor：可以设置触发事务的指定异常，允许指定多个类型的异常。
- noRollbackFor：与 rollbackFor 相反，可以设置不触发事务的指定异常，允许指定多个类型的异常。

4. 事务回滚规则

Spring 的事务回滚通常是根据当前事务抛出异常的时候，Spring 事务管理器捕捉到未经处理的异常，然后根据规则来决定当前事务是否回滚。如果捕获的异常正好是设置 notRollbackFor 属性的异常，那么将不会被捕获。在默认配置下，Spring 只有捕获运行时异常（RuntimeException）的子类时才会进行回滚。

5. @Transactional 使用注意事项

在使用@Transactional 注解的时候，需要注意一些情况：

- @Transactional 需要在类的上方使用，而不是在接口的上方使用，如果在接口上使用，事务就会失效。
- @Transactional 只能在 public 修饰的方法上，如果使用在 private 或 protected 修饰的方法上，事务就会无效。
- @Transactional 尽量不在类的上方使用，因为这样会对类内的全部方法使用事务，如果对查询方法使用事务，就可能会影响效率。

11.6 统一处理异常

11.6.1 异常介绍

异常是程序员的梦魇，甚至对于程序员来说，这是无法避免的事情。当程序在运行的时候发生了一些不被期望的事件，阻止程序按照程序员的预期正常执行，这就是异常。在 Java 语言中提供了异常处理机制来解决异常的出现。

当然，一些异常是可以避免的，也有一些是由于外界因素造成的无法避免的异常，比如：

- 网络波动造成网络通信连接异常。
- 打开不存在的文件。
- 输入非法字符。
- 类型转换异常。
- 对象的空引用。
- JVM 内存溢出。

同时，我们也可以为系统设置业务类异常，当某种场景有悖我们的初衷时，可以定义对应的异常类抛出响应的异常供我们排查。

11.6.2 Java 异常分类

在 Java 标准异常类库中创建了一些通用的异常，这些都是以 Throwable 为顶层父类的，如图 11-8 所示。

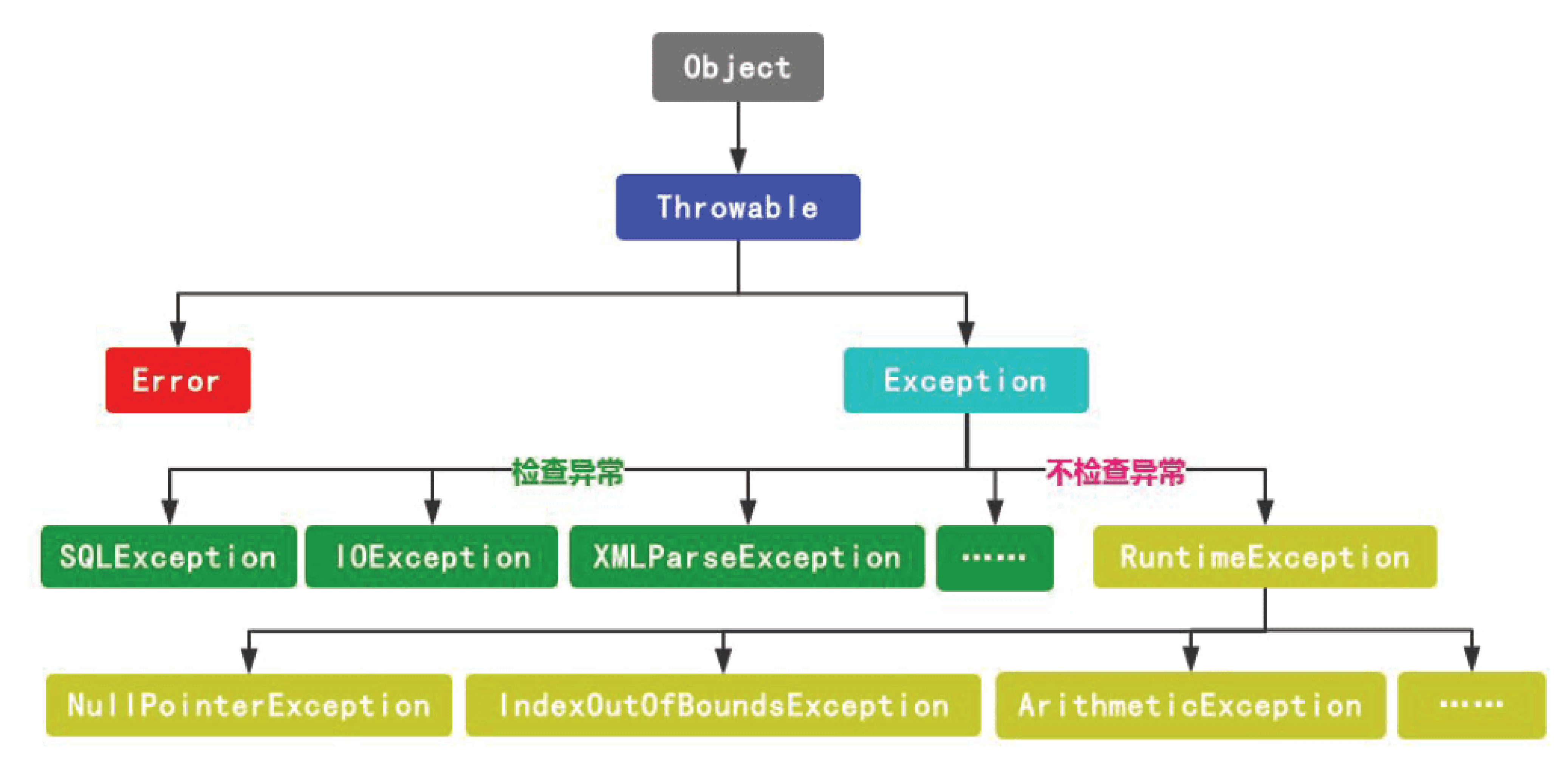


图 11-8　标准异常类结构

Throwable 又派生出 Error 类和 Exception 类。Java 的设计很巧妙，其中：

- Error 类一般代表错误，通常代表 JVM 本身的错误，并且这些错误是无法被修复和捕获的。
- Exception 类代表 Java 标准异常库经常发生的异常。通常代表程序运行时发生的不在预期中的情况，这种异常可以被 Java 异常机制处理。在 Exception 异常中，通常有两个重要的子类，即 IOException 类和 RuntimeException 类。

对于我们常处理的异常，通常分为两类，分别说明如下。

（1）非检查异常（unckecked exception）

Error 和 RuntimeException 以及它们的子类。在 Java 编译的时候，无法发现这样的异常，只有在运行时才能出现该异常。常见的一些非检查异常如下。

- ArithmeticException：当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。
- ArrayIndexOutOfBoundsException：用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，该索引就为非法索引。
- ClassCastException：当试图强制将对象转换为不是实例的子类时，抛出该异常。
- IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数。
- IndexOutOfBoundsException：指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
- NullPointerException：当应用程序试图在需要对象的地方使用 null 时，抛出该异常。
- NumberFormatException：当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。

（2）检查异常（checked exception）

在 Java 编译的过程中，这些错误必须进行修复，如果不对异常进行处理，就无法编译通过。常见的一些检查异常如下。

- ClassNotFoundException：应用程序试图加载类时，找不到相应的类，抛出该异常。
- IllegalAccessException：拒绝访问一个类的时候，抛出该异常。
- NoSuchMethodException：请求的方法不存在。

11.6.3 Spring Boot 中统一处理异常

在 Spring Boot 项目中存在一套异常页面，在访问错误的时候会提示 Whitelabel Error Page。这对于很多互联网项目（网站项目）看起来不是很友好，一般情况下会返回一个统一页面来友好地提示用户。其实在 Spring Boot 中只需要创建一个实现 ErrorController 的类来统一处理发生错误的请求即可，如代码清单 11-20 所示。

代码清单 11-20 Spring Boot-异常项目自定义异常代码

```
@RestController
public class CommonErrorController implements ErrorController {
    private final String ERROR_PATH = "/error";

    @Override
    public String getErrorPath() {
        return ERROR_PATH;
    }

    @RequestMapping(value = ERROR_PATH)
    public String handleError(){
        System.out.println(getErrorPath());
        return "error";
    }
}
```

上述代码可以清晰地看出实现的功能，实质上就是将 error 请求拦截到一个通用方法，这里返回了一串字符串，其实也可以返回一个友好页面展示给用户。具体测试就不展示了，内容比较简单。感兴趣的读者可以设置一些自定义异常进行异常捕获等，快去尝试一下吧。

11.7 使用 AOP

前面介绍了如何使用拦截器打印日志等，其实 Spring 的 AOP 特性可以做到不修改业务的基础上，利用 AOP 对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高开发的效率。本节将介绍 Spring Boot 如何使用 AOP。

11.7.1　AOP 介绍

AOP（Aspect-Oriented Programming，面向切面编程）是 Spring 框架面向切面的编程思想，其利用一种被我们称为“横切”的技术，对 OOP（Object-Oriented Programing，面向对象编程）进行了补充和完善。使用 AOP 的横向切入可以对系统进行无侵入性的日志监听、事务管理、权限管理等。

我们来看一下 AOP 的组成成员。

- Aspect：切面，可以理解为横切多个 class 的一个关注点的模块化。事务管理是一个很好的例子。在 Spring AOP 中，Aspect 可以用普通类或者带有@Aspect 注解的普通类来实现。
- Join point：连接点，程序执行期间的连接点，一般是指方法的调用。
- Advice：通知，在特定切入点上执行的操作。
- Pointcut：切入点，带有通知的连接点。
- Target object：目标对象，由一个或者多个切面通知的对象。
- AOP proxy：AOP 代理，由 AOP 框架创建的对象。

Advice 分为以下几种。

- Before advice：前置通知，在一个连接点之前执行的通知，正常情况下没有办法阻止后面的执行，除非产生异常。
- After returning advice：返回通知，在一个连接点正常执行完所有操作后执行的通知。
- After throwing advice：异常通知，在一个方法抛出异常后执行的通知。
- After(finally) advice：后置通知，在方法结束后执行，无论是正常执行还是异常退出都会执行的通知。
- Around advice：环绕通知，环绕一个连接点，比如方法调用的通知。这是最强的一种通知。环绕通知可以在方法调用之前或之后执行自定义的行为。它也负责选择是否处理连接点方法的执行，通过返回一个特有的值或者抛出异常。环绕通知是使用最普遍的一种通知。

11.7.2　Spring Boot 使用 AOP

接下来，我们以打印日志为例介绍在 Spring Boot 中使用切面的两种方法，即直接使用切面和自定义注解式切面。

1. 直接使用切面

新建项目，在项目中加入 AOP 依赖，pom 文件依赖内容如代码清单 11-21 所示。

代码清单 11-21　Spring Boot-AOP 项目配置文件代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

创建一个日志切面类 LogAspect，在类上加入注解@Aspect，表明这是一个切面类，@Component 表示把当前类实例化到 Spring 容器中，如果含有多个注解，那么可以使用@Order(number)注解指定切面类的优先级（注意：这里的 number 代表数字，数字的值越小，优先级越高），使用这个 LogAspect 切面让 com.dalaoyang.controller 包下所有类打印日志。完整 LogAspect 类内容如代码清单 11-22 所示。

代码清单 11-22　Spring Boot-AOP 项目切面代码

```
@Aspect
@Component
public class LogAspect {
    @Pointcut("execution(public * com.springboot.controller.*.*(..))")
    public void LogAspect(){}

    @Before("LogAspect()")
    public void doBefore(JoinPoint joinPoint){
        System.out.println("doBefore");
    }

    @After("LogAspect()")
    public void doAfter(JoinPoint joinPoint){
        System.out.println("doAfter");
    }

    @AfterReturning("LogAspect()")
    public void doAfterReturning(JoinPoint joinPoint){
        System.out.println("doAfterReturning");
    }

    @AfterThrowing("LogAspect()")
    public void deAfterThrowing(JoinPoint joinPoint){
        System.out.println("deAfterThrowing");
    }

    @Around("LogAspect()")
    public Object deAround(ProceedingJoinPoint joinPoint) throws Throwable{
        System.out.println("deAround");
        return joinPoint.proceed();
    }

}
```

从上述代码清单中可以看到使用了很多注解，分别说明如下。

- @Pointcut：切入点，其中 execution 用于使用切面的连接点。使用方法：execution（方法修饰符（可选）返回类型 方法名 参数 异常模式（可选）），可以使用通配符匹配字符，*可以匹配任意字符。
- @Before：在方法前执行。
- @After：在方法后执行。
- @AfterReturning：在方法执行后返回一个结果后执行。
- @AfterThrowing：在方法执行过程中抛出异常的时候执行。
- @Around：环绕通知，在执行前后都可以使用。这个方法参数必须为 ProceedingJoinPoint，proceed()方法就是被切面的方法，@Before、@After、@AfterReturning 和@AfterThrowing 四个方法可以使用 JoinPoint，JoinPoint 包含类名、被切面的方法名、参数等信息。

直接使用切面的方式大致就是上述这样的，还可以在其中加入业务逻辑，具体可以根据业务需求来使用，接下来笔者带领大家学习如何使用注解式切面。

2. 自定义注解式切面

注解是一种能够被添加到 Java 代码中的元数据，类、方法、变量、参数和包都可以用注解来修饰。注解对于它所修饰的代码并没有直接的影响。创建自定义注解其实和创建接口是一致的。接下来我们创建一个在方法使用前后打印时间的自定义注解。

新建一个自定义注解类 DoneTime，如代码清单 11-23 所示。

代码清单 11-23　Spring Boot-AOP 项目自定义注解代码

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DoneTime {
    String param() default "";
}
```

这样还不算完，还需要为自定义注解创建一个切面，与 11.7.1 小节的切面类一致，这里使用@Around 注解，如代码清单 11-24 所示。

代码清单 11-24　Spring Boot-AOP 项目自定义注解实现类代码

```
@Aspect
@Component
public class DoneTimeAspect {
    @Around("@annotation(doneTime)")
    public Object around(ProceedingJoinPoint joinPoint, DoneTime doneTime)
throws Throwable {
        System.out.println("方法开始时间是:"+new Date());
        Object o = joinPoint.proceed();
```

```
        System.out.println("方法结束时间是:"+new Date()) ;
        return o;
    }
}
```

到这里，两种方式的切面已经创建完成了。接下来，在 com.springboot.controller 包下创建一个 TestController 进行测试，其中 test1 方法加入我们刚刚自定义的注解@DoneTime(param = "TestController")，加入后访问方法，控制台会打印两个切面输出的内容；而 test2 方法由于没有加入自定义注解，因此只会打印 LogAspect 切面日志。

11.8　使用 validator 后台校验

通常来说，在前端提交数据到后端的时候，会进行一定的校验，比如使用 jquery.validate.js 或者当前留下的 Vue 框架中的 vue-validator 进行校验。但是这样还会有一定的风险，所以我们会在后台对数据格式进行校验。在 Java 或 Hibernate 中都提供了一些校验的注解，本节学习使用后台校验数据格式。

首先来看供我们使用的后台校验的注解，分别说明如下。

- @Valid：被注释的元素是一个对象，需要检查此对象的所有字段值。
- @Null：被注释的元素必须为 null。
- @NotNull：被注释的元素必须不为 null。
- @AssertTrue：被注释的元素必须为 true。
- @AssertFalse：被注释的元素必须为 false。
- @Min(value)：被注释的元素必须是一个数字，其值必须大于等于指定的最小值。
- @Max(value)：被注释的元素必须是一个数字，其值必须小于等于指定的最大值。
- @DecimalMin(value)：被注释的元素必须是一个数字，其值必须大于等于指定的最小值。
- @DecimalMax(value)：被注释的元素必须是一个数字，其值必须小于等于指定的最大值。
- @Size(max, min)：被注释的元素的大小必须在指定的范围内。
- @Digits (integer, fraction)：被注释的元素必须是一个数字，其值必须在可接受的范围内。
- @Past：被注释的元素必须是一个过去的日期。
- @Future：被注释的元素必须是一个将来的日期。
- @Pattern(value)：被注释的元素必须符合指定的正则表达式。
- @Email：被注释的元素必须是电子邮箱地址。
- @Length(min=, max=)：被注释的字符串的大小必须在指定的范围内。
- @NotEmpty：被注释的字符串必须非空。
- @Range(min=, max=)：被注释的元素必须在合适的范围内。
- @NotBlank：被注释的字符串必须非空。
- @URL(protocol=, host=, port=, regexp=, flags=)：被注释的字符串必须是一个有效的 URL。

- @CreditCardNumber：被注释的字符串必须通过 Luhn 校验算法，银行卡、信用卡等号码一般都用 Luhn 计算合法性。
- @ScriptAssert (lang=, script=, alias=)：要有 Java Scripting API，即 JSR 223 ("Scripting for the JavaTM Platform")的实现。
- @SafeHtml(whitelistType=,additionalTags=)：classpath 中要有 jsoup 包。

@NotNull、@NotEmpty、@NotBlank 三个注解的区别如下。

- @NotNull：任何对象的 value 不能为 null。
- @NotEmpty：集合对象的元素不为 0，即集合不为空，也可以用于字符串不为 null。
- @NotBlank：只能用于字符串不为 null，并且字符串 trim()以后 length 要大于 0。

接下来，笔者带领大家使用这些注解。新建项目，在 pom 文件中加入 hibernate-validator 依赖，如代码清单 11-25 所示。

代码清单 11-25　Spring Boot-后台校验项目依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.4.Final</version>
</dependency>
```

创建一个 User 实体类，在实体中的几个属性上加入注解进行校验，分别说明如下。

- userName：字段不能为空，并且长度在 6 ~ 12 之间。
- passWord：密码不能为空，并且长度不能小于 6。
- email：需要符合邮箱格式。
- idCard：需要符合身份证格式。
- phone：需要符合手机号格式。

完整 User 实体类内容如代码清单 11-26 所示。

代码清单 11-26　Spring Boot-后台校验项目实体类代码

```
public class User implements Serializable {
    private static final long serialVersionUID = -7362371894429216969L;

    @NotEmpty(message="用户名不能为空")
    @Length(min=6,max = 12,message="用户名长度必须位于 6 到 12 之间")
    private String userName;
```

```
        @NotEmpty(message="密码不能为空")
        @Length(min=6,message="密码长度不能小于 6 位")
        private String passWord;

        @Email(message="邮箱格式错误")
        private String email;

        @Pattern(regexp = "^(\\d{18,18}|\\d{15,15}|(\\d{17,17}[x|X]))$",
message = "身份证格式错误")
        private String idCard;

        @Pattern(regexp = "^((13[0-9]{1})|159|153)+\\d{8}$",message = "手机号格
式错误")
        private String phone;

        ...   //省略 set、Get 方法
    }
```

最后，创建一个 TestController 进行测试。使用 BindingResult 类的 getAllErrors()方法可以获取不符合校验的提示集合，具体怎么展示可以根据项目情况决定，这里是将错误信息拼接成字符串返回前台，如代码清单 11-27 所示。

代码清单 11-27　Spring Boot-后台校验项目测试类代码

```
    @RestController
    public class TestController {
        @PostMapping("/")
        public String testDemo(@Valid User user, BindingResult bindingResult){
            System.out.println(user.toString());
            StringBuffer stringBuffer = new StringBuffer();
            if(bindingResult.hasErrors()){
                List<ObjectError> list =bindingResult.getAllErrors();
                for (ObjectError objectError:list) {
                    stringBuffer.append(objectError.getDefaultMessage());
                    stringBuffer.append("---");
                }
            }
            return stringBuffer!=null?stringBuffer.toString():"";
        }
    }
```

使用 POST 请求访问 testDemo()方法，如果输入不符合校验条件，就会对应输出提示到前台。由于篇幅原因，这里不做测试了，感兴趣的读者可以自行测试。

11.9　使用 Swagger 构建接口文档

在应用开发过程中经常需要对其他应用或者客户端提供 RESTful API 接口，尤其是在版本快速迭代的互联网应用开发中，修改接口的同时还需要同步修改对应的接口文档，这使我们总是做着重复的工作，并且如果忘记修改接口文档，就可能造成不必要的麻烦。本节学习 Spring Boot 框架使用 Swagger 构建 RESTful API。

11.9.1　什么是 Swagger

Swagger（官网地址：https://swagger.io/）是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。总体目标是使客户端和文件系统作为服务器，以同样的速度来更新。文件的方法、参数和模型紧密集成到服务器端的代码中，允许 API 始终保持同步。Swagger 让部署管理和使用功能强大的 API 从未如此简单。

在 Spring Boot 项目中使用 Swagger 其实很简单，大致分为以下三步：

（1）加入 Swagger 依赖。
（2）加入 Swagger 文档配置。
（3）使用 Swagger 注解编写 API 文档和 API 实体模板。

11.9.2　Swagger 2 注解介绍

由于 Swagger 2 提供了非常多的注解供开发使用，这里仅列举一些笔者认为常用的注解。

1. Api

@Api 用在接口文档资源类上，用于标记当前类为 Swagger 的文档资源。其中含有几个常用属性，分别说明如下。

- value：定义当前接口文档的名称。
- description：用于定义当前接口文档的介绍。
- tag：可以使用多个名称来定义文档，但若同时存在 tag 属性和 value 属性，则 value 属性会失效。
- hidden：如果值为 true，就会隐藏文档。

2. ApiOperation

@ApiOperation 用在接口文档的方法上，主要用来注解接口。其中包含几个常用属性，分别说明如下。

- value：对 API 的简短描述。
- note：API 的有关细节描述。

- esponse：接口的返回类型（注意：这里不是返回实际响应，而是返回对象的实际结果）。
- hidden：如果值为 true，就会在文档中隐藏。

3. ApiResponse、ApiResponses

@ApiResponses 和@ApiResponse 二者配合使用返回 HTTP 状态码。@ApiResponses 的 value 值是@ApiResponse 的集合，多个@ApiResponse 用逗号分隔。其中，@ApiResponse 包含的属性如下。

- code：HTTP 状态码。
- message：HTTP 状态信息。
- responseHeaders：HTTP 响应头。

4. ApiParam

@ApiParam 用于方法的参数，其中包含以下几个常用属性。

- name：参数的名称。
- value：参数值。
- required：如果值为 true，就是必传字段。
- defaultValue：参数的默认值。
- type：参数的类型。
- hidden：如果值为 true，就隐藏这个参数。

5. ApiImplicitParam、ApiImplicitParams

二者配合使用在 API 方法上，@ApiImplicitParams 的子集是@ApiImplicitParam 注解，其中@ApiImplicitParam 注解包含以下几个参数。

- name：参数的名称。
- value：参数值。
- required：如果值为 true，就是必传字段。
- defaultValue：参数的默认值。
- dataType：数据的类型。
- hidden：如果值为 true，就隐藏这个参数。
- allowMultiple：是否允许重复。

6. ResponseHeader

API 文档的响应头，如果需要设置响应头，就将@ResponseHeader 设置到@ApiResponse 的 responseHeaders 参数中。@ResponseHeader 提供了以下几个参数。

- name：响应头名称。
- description：响应头备注。

7. ApiModel

设置 API 响应的实体类，用作 API 返回对象。@ApiModel 提供了以下几个参数。

- value：实体类名称。
- description：实体类描述。
- subTypes：子类的类型。

8. ApiModelProperty

设置 API 响应实体的属性，其中包含以下几个参数。

- name：属性名称。
- value：属性值。
- notes：属性的注释。
- dataType：数据的类型。
- required：如果值为 true，就必须传入这个字段。
- hidden：如果值为 true，就隐藏这个字段。
- readOnly：如果值为 true，字段就是只读的。
- allowEmptyValue：如果为 true，就允许为空值。

Swagger 还提供了很多注解，这里就不一一介绍了。如果感兴趣，可以到 Swagger 的 Github 上查找相关文档，地址是 https://github.com/swagger-api/swagger-core。

11.9.3　Spring Boot 使用 Swagger

前面介绍了一些 Swagger 常用的注解。接下来，笔者带领大家学习 Spring Boot 项目使用 Swagger 构建 RESTful API。首先创建一个项目，在项目中加入 Swagger 依赖，如代码清单 11-28 所示。

代码清单 11-28　Spring Boot-Swagger 项目依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
```

```
        <version>2.9.2</version>
    </dependency>
```

接下来创建一个 Swagger 配置类 Swagger2Config，在配置类上加入注解@EnableSwagger2，表明开启 Swagger，注入一个 Docket 类来配置一些 API 相关信息，如 apiInfo() 方法内定义了这样几个文档信息，分别说明如下。

- title：值为接口文档标题。
- description：值为接口文档的详细描述。
- termsOfServiceUrl：一般用于存放公司的地址，这里使用的是 Swagger 官网。
- version：API 文档的版本号。

createRestApi()方法内定义了 Swagger 文档扫描的包，如代码清单 11-29 所示。

代码清单 11-29　Spring Boot-Swagger 项目配置代码

```
    @Configuration
    @EnableSwagger2
    public class Swagger2Config {
        @Bean
        public Docket createRestApi() {
            return new Docket(DocumentationType.SWAGGER_2)
                    .apiInfo(apiInfo())
                    .select()
                    //swagger 文档扫描的包
                    .apis(RequestHandlerSelectors.basePackage("com.springboot.co
ntroller"))
                    .paths(PathSelectors.any())
                    .build();
        }

        private ApiInfo apiInfo() {
            return new ApiInfoBuilder()
                    .title("接口列表 v1.0.0")
                    .description("Swagger2 接口文档地址")
                    .termsOfServiceUrl("https://swagger.io/")
                    .version("v1.0.0")
                    .build();
        }
    }
```

创建一个 User 实体，使用前面介绍的@ApiModel 注解表明这是一个 Swagger 返回的实体，@ApiModelProperty 注解表明几个实体的属性，如代码清单 11-30 所示。

代码清单 11-30　Spring Boot-Swagger 项目实体类代码

```
@ApiModel(value = "用户",description = "用户实体类")
public class User {
    @ApiModelProperty(value = "用户 id",hidden = true)
    private Long id;

    @ApiModelProperty(value = "用户名称")
    private String userName;

    @ApiModelProperty(value = "用户密码")
    private String userPassword;

    ...  //这里省略 set、Get 方法
}
```

最后，创建一个 UserController 作为 API 文档，这个测试 API 没有使用数据库，只是做了一些简单的操作方法供读者查看。结合前面的介绍，相信大家已经对 Swagger 使用有了很深刻的认识。接下来，我们来看一下 UserController 类的完整内容，如代码清单 11-31 所示。

代码清单 11-31　Spring Boot-Swagger 项目 API 文档类代码

```
@RestController
@RequestMapping(value="/users")
@Api(value="用户操作接口",tags={"用户操作接口"})
public class UserController {

    @ApiOperation(value="获取用户详细信息", notes="根据用户的id来获取用户详细信息")
    @ApiImplicitParam(name = "id", value = "用户 ID", required = true,paramType
= "query", dataType = "long")
    @GetMapping(value="/findById")
    public User findById(@RequestParam(value = "id")long id){
        return new User(id,"dalaoyang","123");
    }

    @ApiOperation(value="保存用户", notes="保存用户")
    @PostMapping(value="/saveUser")
    public String saveUser(@RequestBody @ApiParam(name="用户对象",value="传
入 json 格式",required=true) User user){
        return user.toString();
    }

    @ApiOperation(value="修改用户", notes="修改用户")
    @ApiImplicitParams({
            @ApiImplicitParam(name="id",value="主键 id",required=true,
paramType="query",dataType="long"),
```

```
            @ApiImplicitParam(name="username",value="用户名称",
required=true,paramType="query",dataType = "String"),
            @ApiImplicitParam(name="password",value="用户密码",
required=true,paramType="query",dataType = "String")
      })
      @GetMapping(value="/updateUser")
      public String updateUser(@RequestParam(value = "id")long id,
@RequestParam(value = "username")String username,
                        @RequestParam(value = "password")String password){
         User user = new User(id, username, password);
         return user.toString();
      }

      @ApiOperation(value="删除用户", notes="根据用户的 id 来删除用户")
      @ApiImplicitParam(name = "id", value = "用户 ID", required = true,paramType
= "query", dataType = "Integer")
      @ApiResponses({
            @ApiResponse(code = 200,message = "成功！"),
            @ApiResponse(code = 401,message = "未授权！"),
            @ApiResponse(code = 404,message = "页面未找到！"),
            @ApiResponse(code = 403,message = "出错了！")
      })
      @DeleteMapping(value="/deleteUserById")
      public String deleteUserById(@RequestParam(value = "id")int id){
         return "success!";
      }
   }
```

启动项目，访问 http://localhost:8080/swagger-ui.html，可以看到我们定义的文档已经在 Swagger 页面上显示了，如图 11-9 所示。

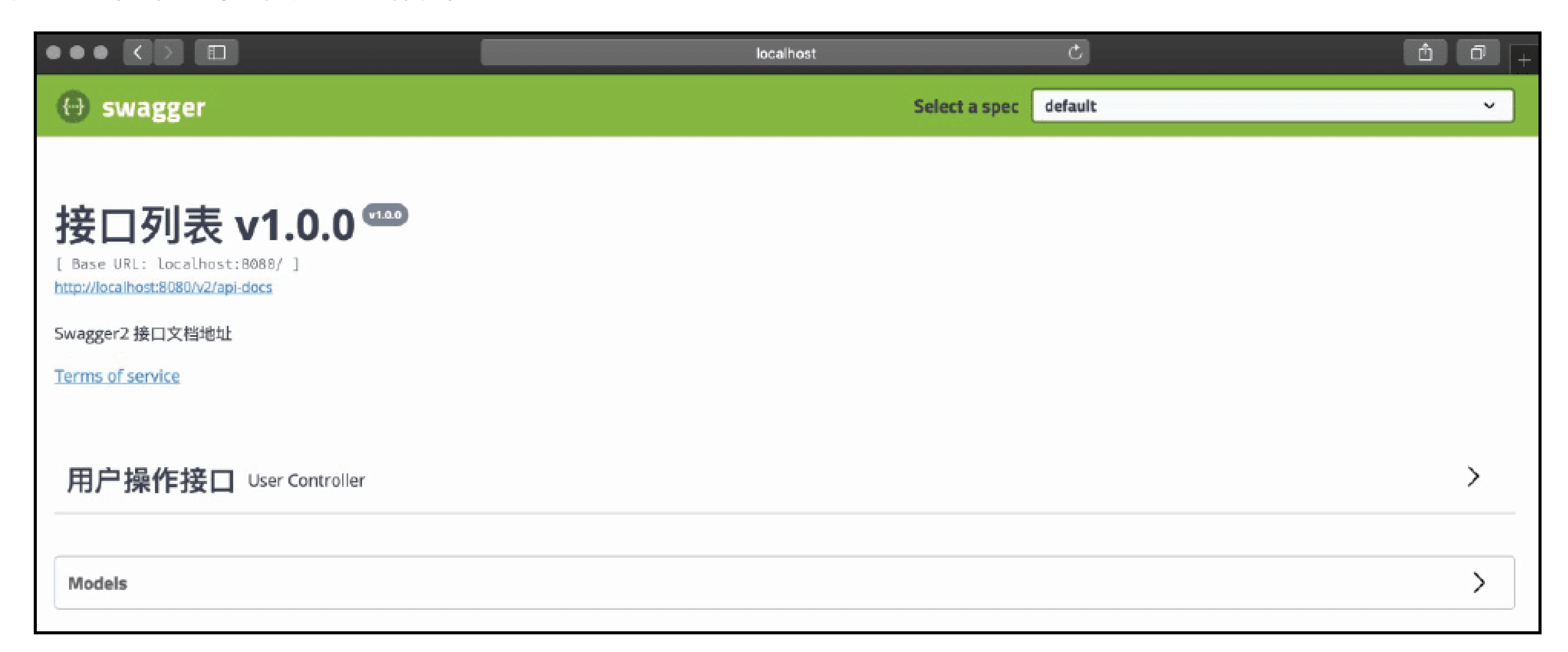


图 11-9　Swagger 项目文档页面首页

单击用户操作接口，可以看到这个 API 中的几个具体的接口，如图 11-10 所示。

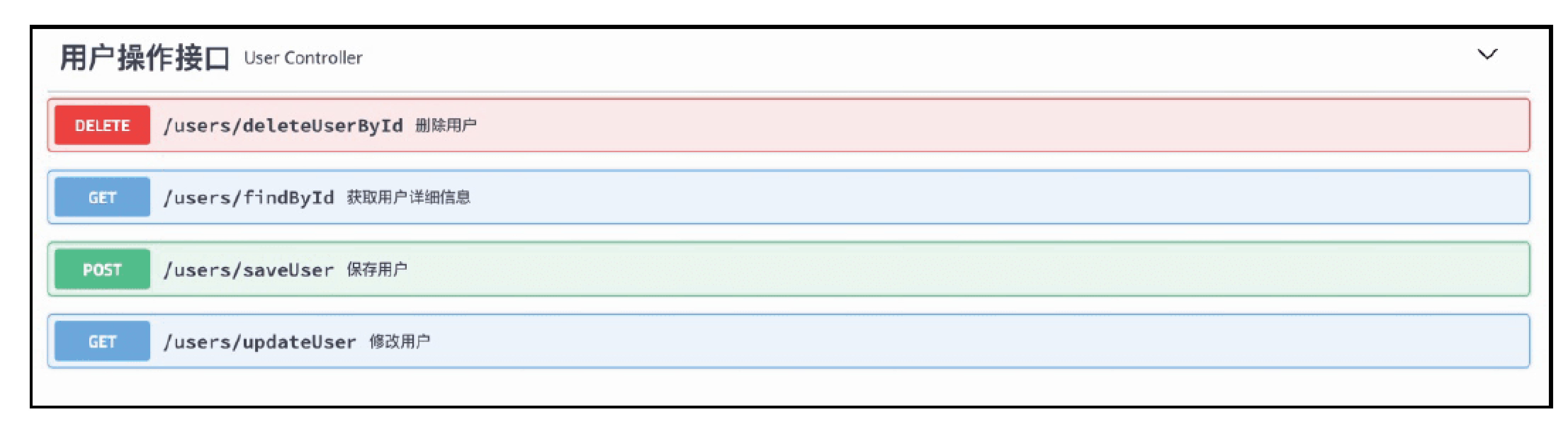


图 11-10　Swagger 项目 API 图片

单击下方的 Models，可以查看我们在项目内定义的接口返回对象列表。这里只定义了一个 User 实体，如图 11-11 所示。

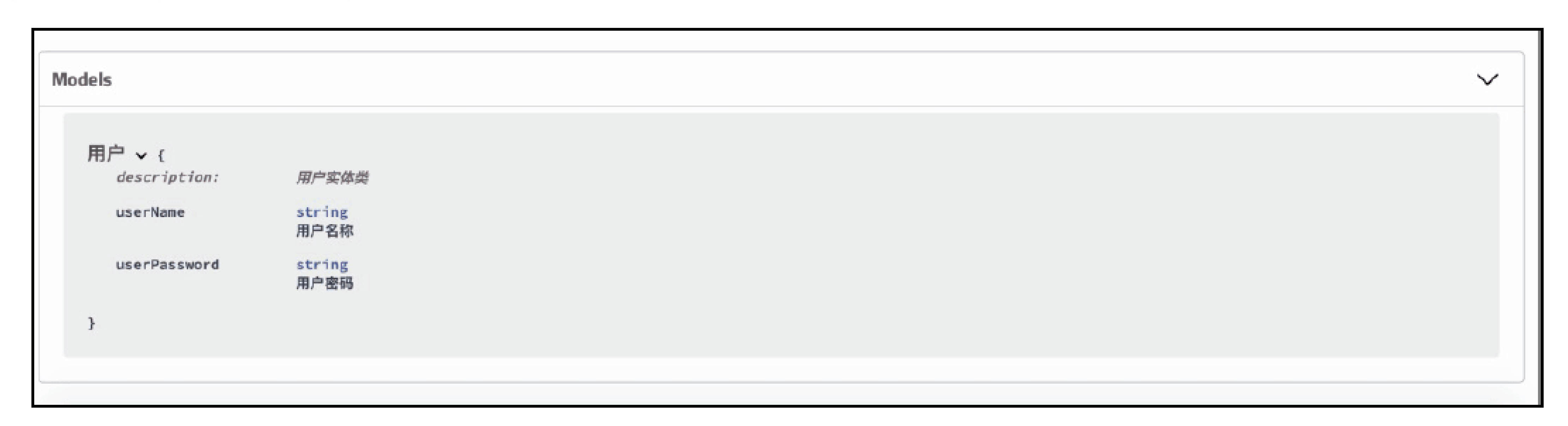


图 11-11　Swagger 项目 Models 实体

回到接口列表，以删除用户接口为例，单击删除用户接口，可以看到接口定义的参数、返回值、响应码等，单击 Try it out 按钮可以发起调用请求、删除用户接口，如图 11-12 所示。

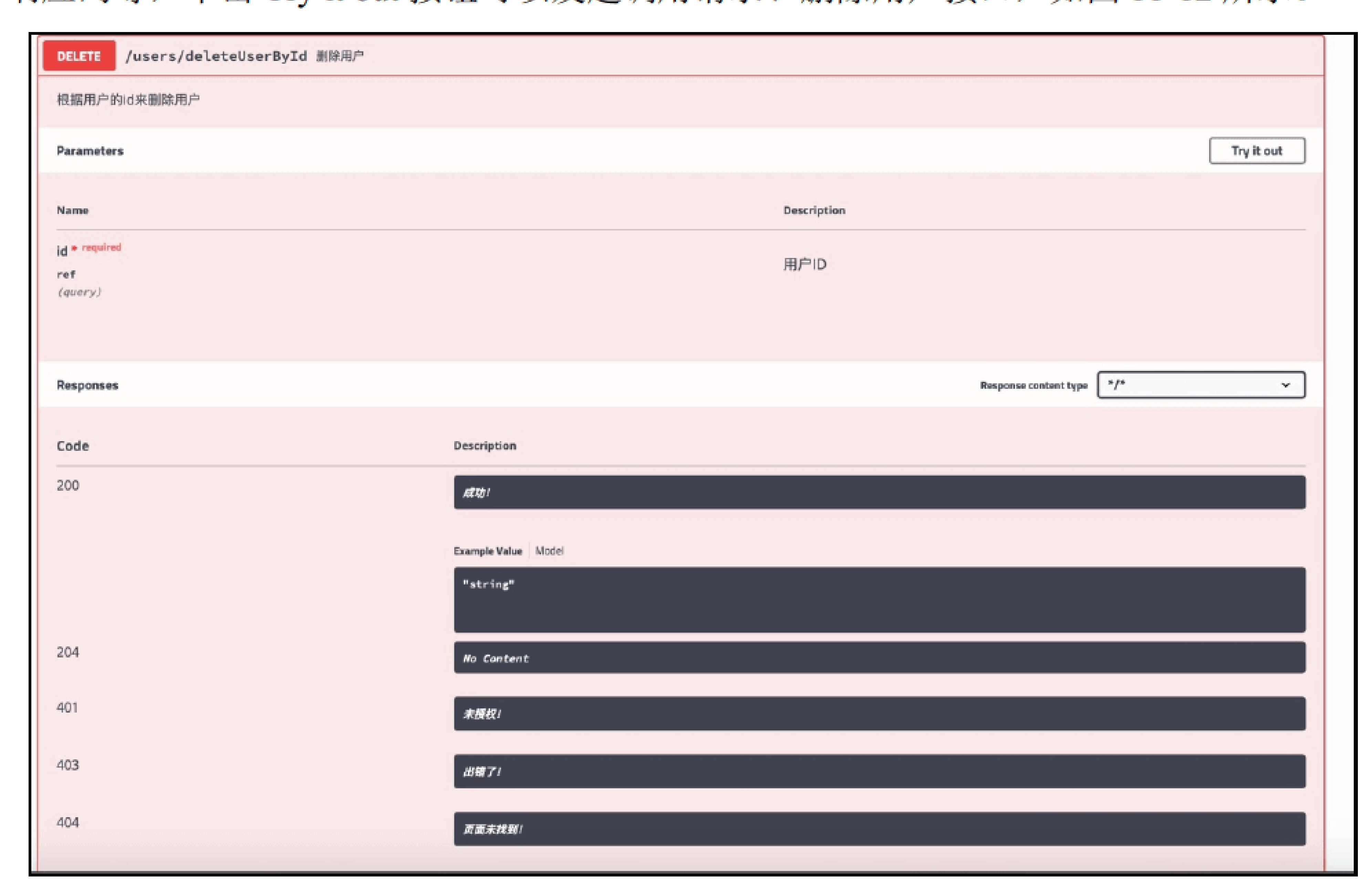


图 11-12　Swagger 项目删除用户文档代码

至于其他接口，这里就不展示了，感兴趣的读者可以下载示例代码进行查看。

11.10 使用 ApiDoc 构建接口文档

11.9 节介绍了使用 Swagger 构建接口文档，本节介绍另一个构建接口文档的框架——ApiDoc 框架。

11.10.1 如何使用 ApiDoc 接口文档

在 Spring Boot 应用程序中使用 ApiDoc 文档很简单，大致分为如下三步：

（1）加入 ApiDoc 依赖。
（2）在启动类上加入@EnableApi2Doc 注解启用 ApiDoc。
（3）构建 ApiDoc 文档和实体。

11.10.2 ApiDoc 常用注解

ApiDoc 中没有提供像 Swagger 那么多的注解，ApiDoc 的注解都是可以在类上、实体上共同使用的。ApiDoc 提供的常用注解如下。

1. Api2Doc

@Api2Doc 注解主要用于对文档的生成。如果@Api2Doc 修饰在类上，就相当于将当前类作为 ApiDoc 文档。当服务启动时，Api2Doc 会扫描 Spring 容器中所有的 Controller 类，当 Controller 类上含有@Api2Doc 注解时，才会被 Api2Doc 生成接口文档。当生成接口文档后，会在 ApiDoc 页面左侧生成一个菜单，其中注解内的 name 值就是菜单名。同时，@Api2Doc 注解可以修饰在方法中。@Api2Doc 注解提供了以下常用参数。

- name：定义名称。
- order：排序优先级，数字越小，越靠前。

2. ApiComment

@ApiComment 注解主要用于对 API 进行说明，它可以修饰在很多地方。当修饰在类上时，表示对当前 API 接口进行说明；当修饰在方法上时，表示对这个 API 接口进行说明；当修饰在参数上时，表示对这个 API 接口的请求参数进行说明；当修饰在返回类型的属性上时，表示对这个 API 接口的返回字段进行说明；当修饰在枚举项上时，表示对枚举项进行说明。@ApiComment 注解提供了如下参数。

- value：说明。
- seeField：采用指定字段上的说明信息。
- seeClass：采用指定类的同名字段上的说明信息。
- sample：示例值。

3. ApiError

@ApiError 用于定义错误码和错误说明，包含的属性如下。

- value：定义错误码。
- comment：定义错误说明。

4. ApiErrors

用于组装错误集合，配合@ApiError 注解使用。

11.10.3　Spring Boot 使用 ApiDoc

接下来，以实际项目为例使用 ApiDoc。新建项目，在项目中加入 terran4j-commons-api2doc 依赖，如代码清单 11-32 所示。

代码清单 11-32　Spring Boot-APIDOC 项目依赖代码

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>com.github.terran4j</groupId>
    <artifactId>terran4j-commons-api2doc</artifactId>
    <version>1.0.2</version>
</dependency>
```

在启动类上加入@EnableApi2Doc 注解，如代码清单 11-33 所示。

代码清单 11-33　Spring Boot-APIDOC 项目启动类代码

```
@SpringBootApplication
@EnableApi2Doc
public class Chapter1110Application {
    public static void main(String[] args) {
        SpringApplication.run(Chapter1110Application.class, args);
    }
}
```

创建实体类用作 API 返回对象，如代码清单 11-34 所示。

代码清单 11-34　Spring Boot-APIDOC 项目实体类代码

```
public class User {
    @ApiComment(value = "用户 id", sample = "1")
    private Long id;
```

```
    @ApiComment(value = "用户 id", sample = "dalaoyang")
    private String userName;

    @ApiComment(value = "用户 id", sample = "123")
    private String userPassword;

    ... //省略 set，Get 方法
}
```

最后，创建一个 UserController 作为 API，使用的都是简单逻辑，这里就不做介绍了。UserController 内容如代码清单 11-35 所示。

代码清单 11-35 Spring Boot-APIDOC 项目 API 文档代码

```
@Api2Doc(id = "users", name = "用户接口", order = 1)
@ApiComment(seeClass = User.class)
@RestController
@RequestMapping(value = "/api/v1/users")
public class UserController {
    @Api2Doc(order = 1)
    @ApiComment("新增用户")
    @ApiErrors({
            @ApiError(value = "is-exists", comment = "此用户已经存在！"),
            @ApiError(value = "error", comment = "错误！")
    })
    @PostMapping(name = "新增用户",value="/addUser")
    public User addUser(
            @ApiComment(value = "用户名称",seeField = "id") @RequestParam
(required = true) Long id,
            @ApiComment(value = "用户名称",seeField = "userName") @RequestParam
(required = true) String userName,
            @ApiComment(value = "用户密码",seeField = "userPassword")
@RequestParam(required = true) String userPassword) {
        User user = new User(id,userName,userPassword);
        return user;
    }

    @Api2Doc(order = 2)
    @ApiComment("根据用户 ID 查询用户")
    @ApiError(value = "not-found", comment = "此用户不存在！")
    @GetMapping(name = "查询用户", value = "/getUser/{id}")
    public User getUser(@PathVariable("id") Long id) {
        return new User(id,"dalaoyang","123");
    }
}
```

启动项目，访问 http://localhost:8080/api2doc/home.html，可以看到如图 11-13 所示的页面。



图 11-13　ApiDoc 项目首页

单击左侧菜单栏的用户接口，可以看到如图 11-14 所示的页面。



图 11-14　ApiDoc 项目新增用户文档页面 1

在接口详情下会有一些调用的详细信息，如图 11-15 所示。

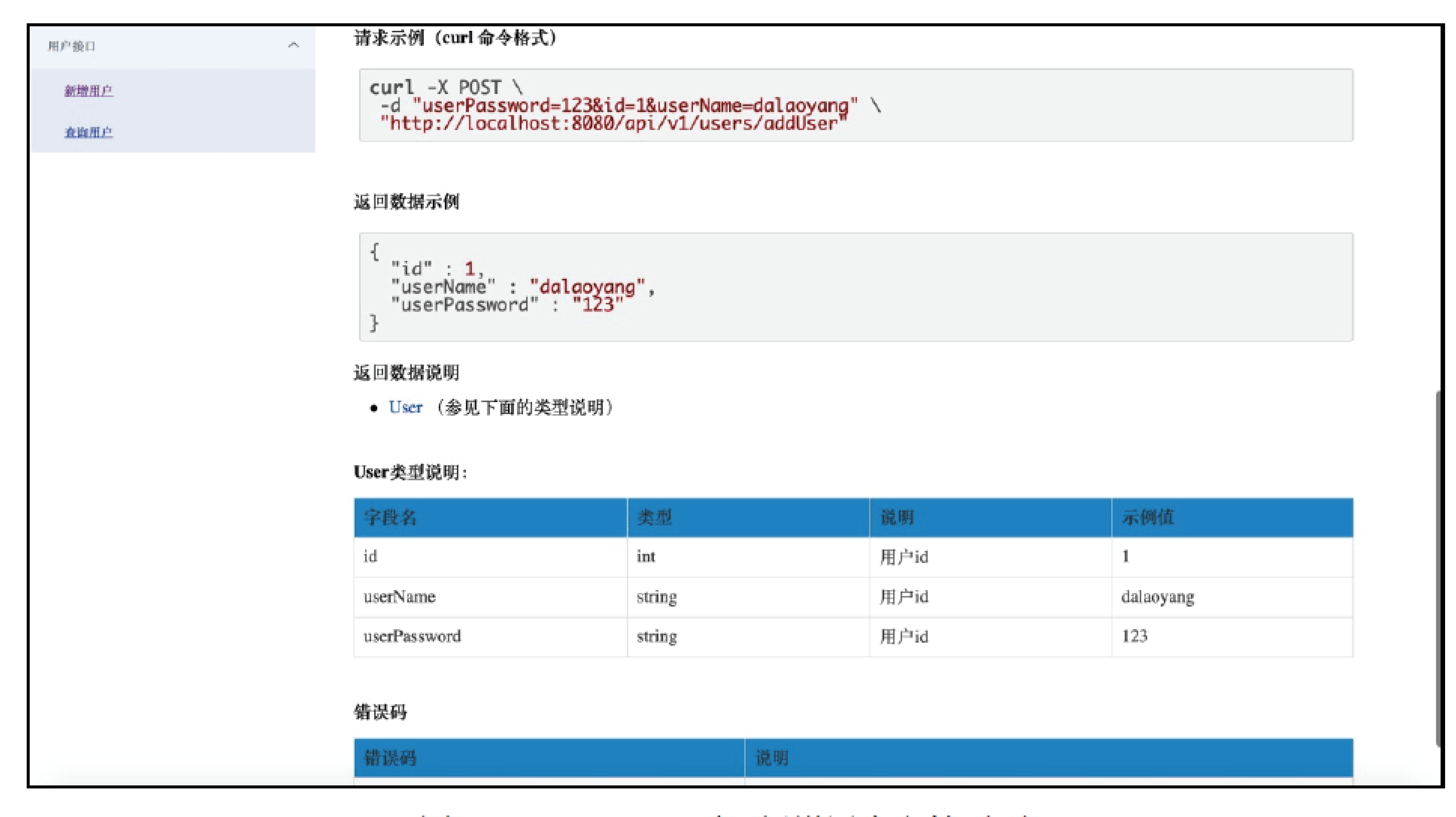


图 11-15　ApiDoc 项目新增用户文档页面 2

ApiDoc 文档相较于 Swagger 美中不足的是没有直接调用的地方，只有一个文档供我们查看，至于项目中使用哪个，根据实际情况选择即可。

11.11 小　结

本章的内容可能比之前几个章节零散，但是这些功能基本上都是项目中必须涉及的，所以更需要我们了解其使用方法和功能，从而在项目中使用。

第 12 章

Spring Boot 打包部署

在之前的章节中，对 Spring Boot 应用程序与其他常用工具进行了整合，从而创建了一个完整的应用程序。本章将对 Spring Boot 应用的几种运行和部署进行介绍，让应用可以在本机或服务器上顺利运行。

12.1 使用 IDE 启动

关于 Spring Boot 应用程序的启动方式有很多种，对于开发者来说，常用的是使用 IDE 启动 Spring Boot 应用程序。由于这种方式比较简单，因此以 IntelliJ IDEA 为例简单介绍一下如何在 IDE 上运行 Spring Boot 应用程序。

在 IDE 上运行 Spring Boot 应用程序主要分为两种模式，即 Run 模式和 Debug 模式。其中，Run 模式是直接运行应用代码；Debug 模式是以调试模式运行代码，可以进行 Debug 调试。接下来分别介绍两种模式的使用。

12.1.1 运行 Spring Boot 应用程序

这里以 8.1 节的项目为例，在 IntelliJ IDEA 中，可以直接单击快捷栏的 Run 按钮或者 Debug 按钮来启动，如图 12-1 所示。

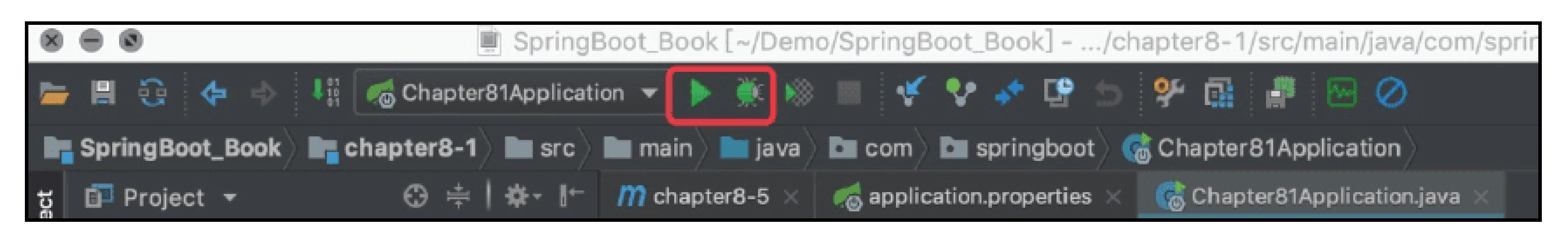


图 12-1　IntelliJ IDEA 启动应用按钮

单击按钮后，即可正常运行 Spring Boot 应用程序。当然，我们也可以在 Spring Boot 应用程序主类（Application 类）处右击，然后选择 Run 模式或者 Debug 模式，运行 main 方法来启动应用程序，如图 12-2 所示。



图 12-2　在启动类执行 main 方法启动程序

上述两种模式可以实现同样的效果，在开发中结合自己的喜好来使用即可。

12.1.2　IntelliJ IDEA 启动多实例

在开发测试功能的过程中，可能有很多时候需要相同的项目以不同的端口启动。这时，有的开发人员可能选择复制项目，然后分别启动。其实在 IntelliJ IDEA 中提供了单实例多端口启动功能，这里还是以 8.1 节的项目为例，首先启动项目，当前端口为 8080，然后在 IntelliJ IDEA 中单击 Toolbar 上的编辑项目按钮，单击后选择 Edit Configurations 选项，如图 12-3 所示。



图 12-3　Edit Configurations 选项

打开如图 12-4 所示的页面，取消选中 Single instance only 复选框。

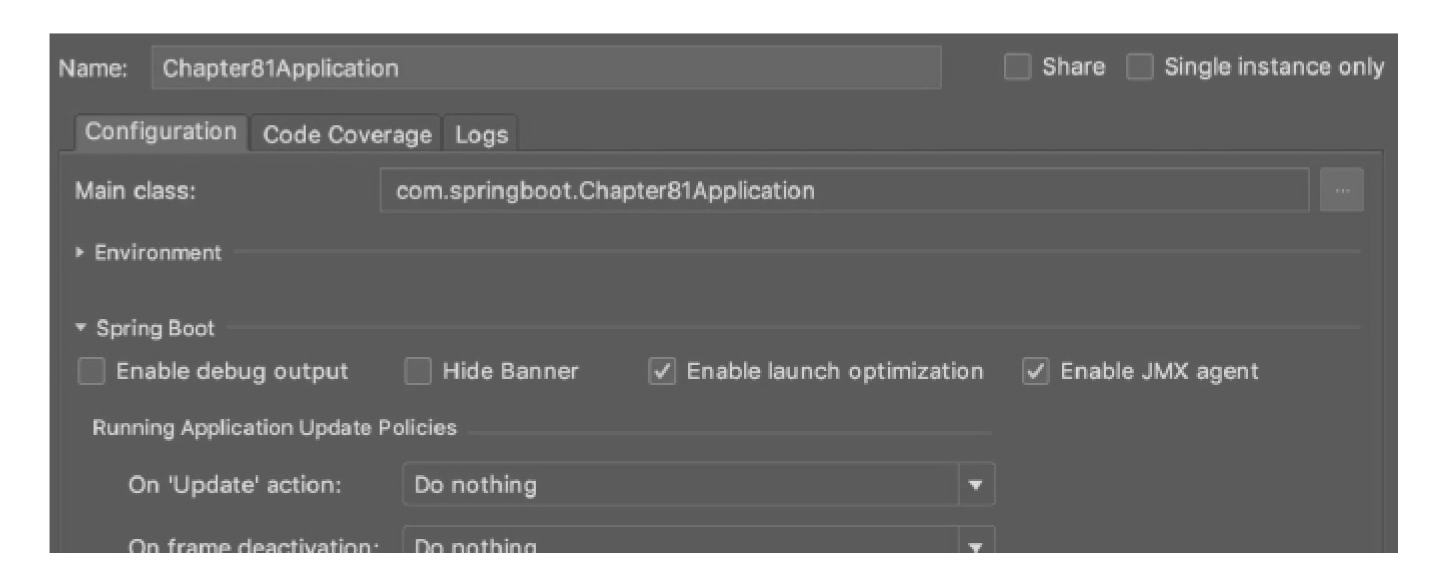


图 12-4　取消选中 Single instance only 复选框

然后修改配置文件端口，继续启动 Spring Boot 应用程序即可实现多端口启动。

12.2　使用 Maven 启动

Maven 是本书用来构建 Spring Boot 的主要工具，其实 Spring Boot 应用也可以使用 Maven 命令来构建。命令如代码清单 12-1 所示。

代码清单 12-1　Maven 启动 Spring Boot 应用程序

```
mvn spring-boot:run
```

使用命令的时候需要在 Spring Boot 应用程序根目录，这里以 8-1 节的项目为例，首先在终端（命令行）进入项目根目录，如图 12-5 所示。

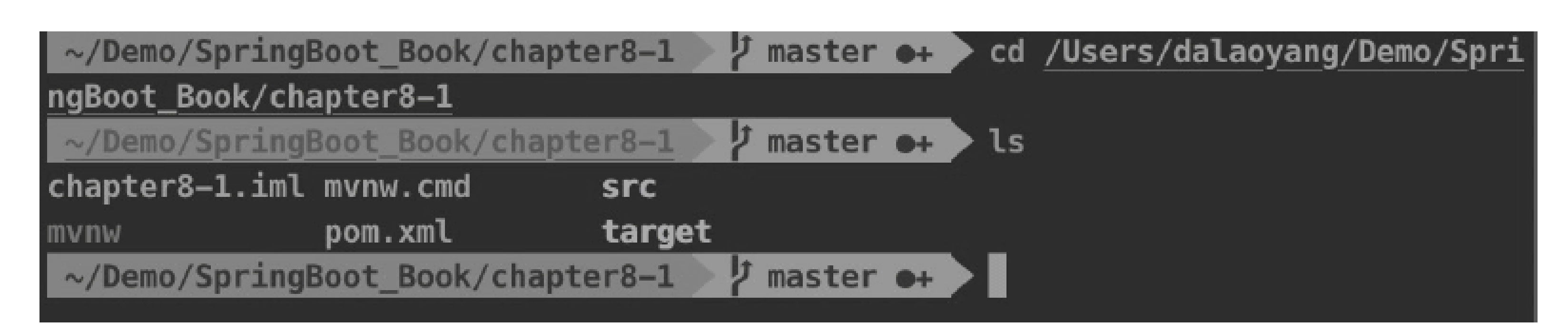


图 12-5　项目文件目录

然后在当前文件夹下指定代码清单 12-1 的命令，就可以启动 Spring Boot 应用程序（注意：需要配置 Maven 环境变量）。

12.3　JAR 形式启动

JAR 形式运行和部署 Spring Boot 应用是 Spring Boot 一个比较大的亮点，其内嵌的 Web 容器起着重要的作用。接下来，介绍 Spring Boot 应用程序如何以 JAR 包形式启动。

12.3.1　使用命令将 Spring Boot 应用程序打成 JAR

使用 12.2 节的目录，分别利用以下 3 个命令进行打包。

- mvn clean package：命令完成了项目编译、单元测试、打包。
- mvn clean install：命令不但完成了项目编译、单元测试、打包，还将打好的 JAR 包部署到本地 Maven 仓库。
- mvn clean depoly：命令不但完成了项目编译、单元测试、打包，还将打好的 JAR 包部署到本地 Maven 仓库和远程 Maven 私服仓库。

在使用命令打好 JAR 包之后，在目录下会生成一个 target 文件夹，进入文件夹后可以看到打好的 JAR 包，如图 12-6 所示。

```
1. dalaoyang@dalaoyang-3: ~/Demo/SpringBoot_Book/chapter8-
~/Demo/SpringBoot_Book/chapter8-1/target  master ●+  ll
total 37328
-rw-r--r--  1 dalaoyang  staff   18M 12 15 23:54 chapter8-1-0.0.1-SNAPSHOT.jar
-rw-r--r--  1 dalaoyang  staff  6.0K 12 15 23:54 chapter8-1-0.0.1-SNAPSHOT.jar.original
drwxr-xr-x  4 dalaoyang  staff  128B 12 15 23:54 classes
drwxr-xr-x  3 dalaoyang  staff   96B 12 15 23:54 generated-sources
drwxr-xr-x  3 dalaoyang  staff   96B 12 15 23:54 generated-test-sources
drwxr-xr-x  3 dalaoyang  staff   96B 12 15 23:54 maven-archiver
drwxr-xr-x  3 dalaoyang  staff   96B 12 15 23:54 maven-status
drwxr-xr-x  4 dalaoyang  staff  128B 12 15 23:54 surefire-reports
drwxr-xr-x  3 dalaoyang  staff   96B 12 15 23:54 test-classes
~/Demo/SpringBoot_Book/chapter8-1/target  master ●+
```

图 12-6　项目文件 JAR 文件目录

其中，chapter8-1-0.0.1-SNAPSHOT.jar 就是这里需要使用的 JAR 包文件，然后以 java -jar JAR 文件名称的形式启动 JAR 包，具体启动命令如代码清单 12-2 所示。

代码清单 12-2　命令行启动 Spring Boot 应用程序

```
java -jar chapter8-1-0.0.1-SNAPSHOT.jar
```

执行命令后即可正确启动 Spring Boot 应用程序。

当然，Spring Boot 应用程序 JAR 形式启动可以指定一些参数，分别说明如下。

指定端口：java -jar xxx.jar --server.port=8888。

内存参数：java -Xms800m -Xmx800m -XX:PermSize=256m -XX:MaxPermSize=512m -XX:MaxNewSize=512m -jar xxx.jar。

配置文件：java -jar xxx.jar -Dspring.profiles.active=dev。

后台运行：nohup java -jar xxx.jar &。

常用命令还有很多，这里只列出了一小部分，仅供参考。

12.3.2　IntelliJ IDEA 打 JAR 包

在 IntelliJ IDEA 内打 JAR 包比较简单，直接在工具栏 Maven Projects 对应项目单击 package、install 或 deploy 按钮即可（例如，单击 package 按钮的效果等同于执行 mvn package 命令），如图 12-7 所示。

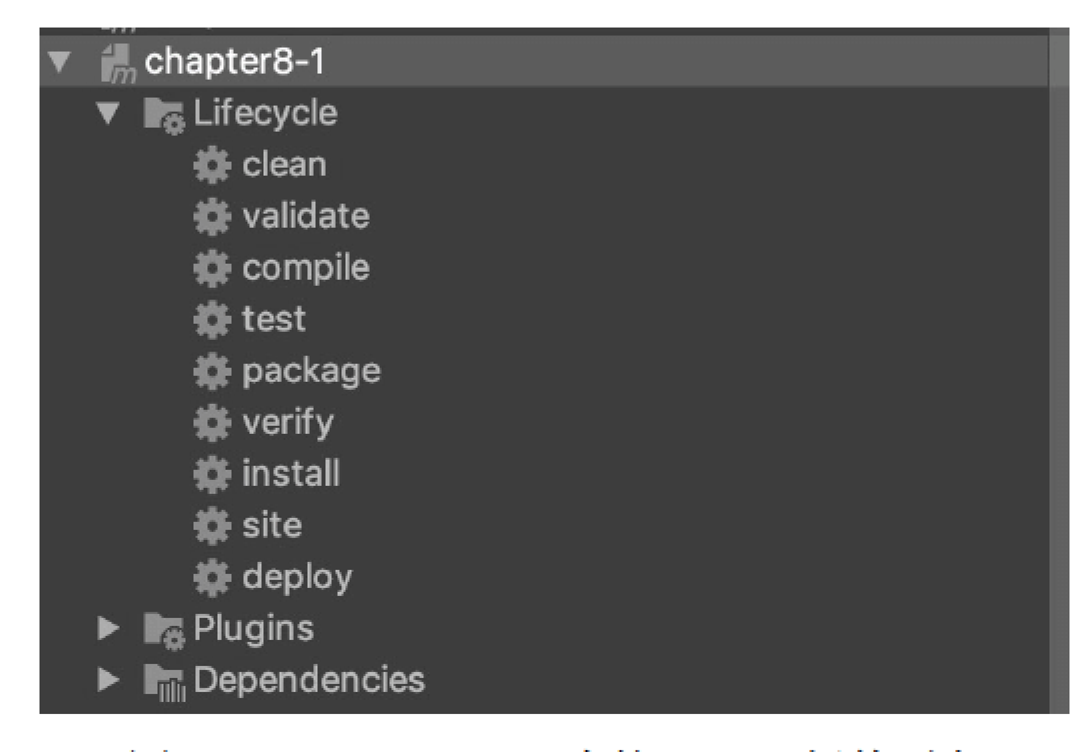


图 12-7　IntelliJ IDEA 内的 Maven 插件列表

单击对应按钮后，后续操作还是使用 java -jar JAR 文件名称来执行应用程序。

12.4　War 形式启动

传统的 Java Web 应用程序一般都是在应用开发完成之后将应用程序打包成 War 包，然后部署到 Tomcat、WebLogic 等 Web 容器下。对于 Spring Boot 应用程序来说，虽然提供了 JAR 形式的部署，但是也支持使用 War 包部署。接下来，笔者将带领大家学习如何使用 War 部署 Spring Boot 应用程序。

12.4.1　创建项目

新建一个 Spring Boot 项目，因为 Spring Boot 应用程序默认是使用 Tomcat 容器的，而我们这次需要打 War 包的形式部署到 Tomcat，所以需要在 pom 文件中移除 spring-boot-starter-tomcat 依赖，并且加入 spring-boot-starter-web 依赖，便于我们进行测试，如代码清单 12-3 所示。

代码清单 12-3　Spring Boot 应用程序 War 部署依赖文件代码

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

接下来，创建一个 ServletInitializer 类，继承 SpringBootServletInitializer 类来实现 configure 方法，如代码清单 12-4 所示。

代码清单 12-4　Spring Boot 应用程序 War 部署 ServletInitializer 类代码

```
public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(Chapter114Application.class);
    }

}
```

其实，到这里就已经配置完成了。不过我们需要创建一个 Controller 进行测试，这里创建一个 IndexController，里面写一个简单的返回字符串便于测试，如代码清单 12-5 所示。

代码清单 12-5　Spring Boot 应用程序 War 部署 IndexController 类代码

```
@RestController
public class IndexController {

    @GetMapping("/")
    public String index(){
        return "SpringBoot War Application !";
    }
}
```

12.4.2　打 War 包部署到 Tomcat

配置完成后，使用 12.3 节打 JAR 包的方法将项目打成 WAR 包。这里以终端执行为例，执行 mvn clean install 后，查看项目内的 target 文件夹，如图 12-8 所示。

```
~/Demo/SpringBoot_Book/chapter11-4/target  master ●+  ll
total 52992
drwxr-xr-x  4 dalaoyang  staff   128B 12 16 00:59 chapter11-4-0.0.1-SNAPSHOT
-rw-r--r--  1 dalaoyang  staff    15M 12 16 00:59 chapter11-4-0.0.1-SNAPSHOT.war
-rw-r--r--  1 dalaoyang  staff    10M 12 16 00:59 chapter11-4-0.0.1-SNAPSHOT.war.original
drwxr-xr-x  4 dalaoyang  staff   128B 12 16 00:58 classes
drwxr-xr-x  3 dalaoyang  staff    96B 12 16 00:58 generated-sources
drwxr-xr-x  3 dalaoyang  staff    96B 12 16 00:58 generated-test-sources
drwxr-xr-x  3 dalaoyang  staff    96B 12 16 00:59 maven-archiver
drwxr-xr-x  3 dalaoyang  staff    96B 12 16 00:58 maven-status
drwxr-xr-x  4 dalaoyang  staff   128B 12 16 00:59 surefire-reports
drwxr-xr-x  3 dalaoyang  staff    96B 12 16 00:58 test-classes
```

图 12-8　项目文件 War 文件目录

接下来，我们把 chapter11-4-0.0.1-SNAPSHOT.war 文件放入 Tomcat 应用的 webapps 文件夹下，如图 12-9 所示。

```
~/tomcat/apache-tomcat-8.5.24-springboot/webapps  ll
total 31536
drwxr-x---   10 dalaoyang  staff   320B  3 15  2018 ROOT
-rw-r--r--    1 dalaoyang  staff    15M 12 16 00:59 chapter11-4-0.0.1-SNAPSHOT.war
drwxr-xr-x@ 55 dalaoyang  staff   1.7K 12 29  2017 docs
drwxr-xr-x@  7 dalaoyang  staff   224B 12 29  2017 examples
drwxr-xr-x@  7 dalaoyang  staff   224B 12 29  2017 host-manager
drwxr-xr-x@  8 dalaoyang  staff   256B 12 29  2017 manager
```

图 12-9　Tomcat 文件夹 webapps 文件目录

最后，进入 Tomcat 应用 bin 目录执行命令./startup.sh 来启动 Tomcat。关于测试，这里就不演示了，感兴趣的读者可以自行测试。

12.5　使用 Docker 构建 Spring Boot 项目

现如今很多公司部署服务都使用容器部署，而 Docker 正是这一领域的佼佼者。接下来笔者将介绍如何使用 Docker 部署 Spring Boot 应用。

12.5.1　Docker 简介

Docker（官网地址：https://www.docker.com/）是一个开源的应用容器引擎，让开发者可以打包其应用及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上。也可以实现虚拟化，容器是完全使用沙箱机制的，相互之间不会有任何接口。

Docker 基于 Linux 内核的 CGroup、Namespace 技术，对进程进行封装隔离，可以使 Linux 服务器最大化地发挥性能。接下来我们看一下如何利用 Docker 构建 Spring Boot 应用程序。

12.5.2　安装 Docker

首先需要有一个 Docker 环境。安装 Docker 很简单，这里以 mac 为例，在终端执行命令，如代码清单 12-6 所示。

代码清单 12-6　Mac 安装 Docker 命令

```
brew cask install docker
```

安装完成后，在应用程序内启动 Docker.app 即可。

12.5.3　Dockerfile

使用 Docker 构建 Spring Boot 应用程序需要创建一个 Dockerfile 文件（没有后缀）。这里以 8-1 节的项目为例，进入 target 文件夹创建 Dockerfile 文件，内容如代码清单 12-7 所示。

代码清单 12-7　Dockerfile 文件内容

```
FROM java:8
VOLUME /tmp
ADD chapter8-1-0.0.1-SNAPSHOT.jar /test.jar
ENTRYPOINT
["java","-Djava.security.egd=file:/dev/./urandom","-jar","/test.jar"]
```

下面对 Dockerfile 进行简单介绍。

第 1 行：java:8 使用 jdk 版本。
第 2 行：临时文件目录。
第 3 行：chapter8-1-0.0.1-SNAPSHOT.jar 是 JAR 文件内容。
第 4 行：ENTRYPOINT 执行 JAR 文件 12.5.4 生成 Docker 镜像。

接下来，通过命令生成 Docker 镜像，如代码清单 12-8 所示。

代码清单 12-8　Dockerfile 文件内容

```
docker build -t test .
```

构建 Docker 镜像 test，执行后如图 12-10 所示，就执行成功了。

```
✗ ~/Demo/SpringBoot_Book/chapter8-1/target  master ●+  docker build -t test .
Sending build context to Docker daemon  19.16MB
Step 1/4 : FROM java:8
8: Pulling from library/java
5040bd298390: Pull complete
fce5728aad85: Pull complete
76610ec20bf5: Pull complete
60170fec2151: Pull complete
e98f73de8f0d: Pull complete
11f7af24ed9c: Pull complete
49e2d6393f32: Pull complete
bb9cdec9c7f3: Pull complete
Digest: sha256:34959eed5df6bfe74a17fbf459ed29505d51a48d5cb4a39f215b613c65b8023b
Status: Downloaded newer image for java:8
 ---> d23bdf5b1b1b
Step 2/4 : VOLUME /tmp
 ---> Running in 3c63fa8d74a5
Removing intermediate container 3c63fa8d74a5
 ---> 5771201bac6f
Step 3/4 : ADD chapter8-1-0.0.1-SNAPSHOT.jar /test.jar
 ---> 2efb8c1baa39
Step 4/4 : ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/test.jar"]
 ---> Running in f5363ba4424b
Removing intermediate container f5363ba4424b
 ---> db7f94e2fe37
Successfully built db7f94e2fe37
Successfully tagged test:latest
```

图 12-10　docker build 执行结果

当然，也可以通过 docker images 查看是否生成 Docker 镜像，查询结果如图 12-11 所示。

```
~/Demo/SpringBoot_Book/chapter8-1/target  master ●+  docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test                latest              db7f94e2fe37        6 minutes ago       662MB
java                8                   d23bdf5b1b1b        23 months ago       643MB
```

图 12-11　docker images 执行结果

12.5.4　运行 Docker 镜像

执行 Docker 镜像只需要执行 docker run 命令即可，其中可以通过-d 来指定后台运行，-p 来指定容器内端口以及服务器端口，最后指定运行的镜像名称。完整运行 Docker 镜像的代码如代码清单 12-9 所示。

代码清单 12-9　运行 Docker 镜像代码

```
docker run -d -p 8080:8080 test
```

运行后，在浏览器访问 http://localhost:8080/actuator/health，可以看到如下结果：

```
{"status":"DOWN","details":{"my":{"status":"DOWN","details":{"Error Code:":500}},"diskSpace":{"status":"UP","details":{"total":62725623808,"free":57663266816,"threshold":10485760}}}
```

到这里，使用 Docker 运行 Spring Boot 应用就完成了。更多 Docker 命令可以在官网中查看。

12.6　使用 Jenkins 自动化部署 Spring Boot 应用

当服务逐渐扩张，部署变成了一道难题，自动化部署随之流行。Jenkins 是自动集成部署项目的优秀产品，本节将介绍 Spring Boot 如何通过 Jenkins 进行自动部署。

12.6.1　Jenkins 简介

Jenkins（官网地址：https://jenkins.io/）是一个由 Java 语言开发的持续集成交付的项目。无论是什么语言构建的项目，几乎都可以通过 Jenkins 部署，并且 Jenkins 的安装非常简单，只要是拥有 Java 环境的服务器或者开发环境，都可以通过下载 WAR 的形式直接启动运行。

12.6.2　Spring Boot 应用使用 Jenkins

本节介绍 Jenkins 从 Git 平台拉取项目代码，然后将项目构建成 JAR 包文件，最后通过执行 Shell 来启动 Spring Boot 应用程序。这里使用的 Git 项目是托管在码云上的项目，可以免费供大家使用，页面地址是 https://gitee.com/dalaoyang/Test-Jenkins，Git 地址是 https://gitee.com/dalaoyang/Test-Jenkins。

1. 安装 Jenkins

直接通过 Jenkins 官网（下载地址：https://jenkins.io/download/）下载最新版本的 Jenkins，然后选择喜欢的方式安装即可。

2. Jenkins 插件

这里使用 Jenkins 的 Maven 插件和 Git 插件，在“系统管理”→“插件管理”处下载即可，如图 12-12 所示。

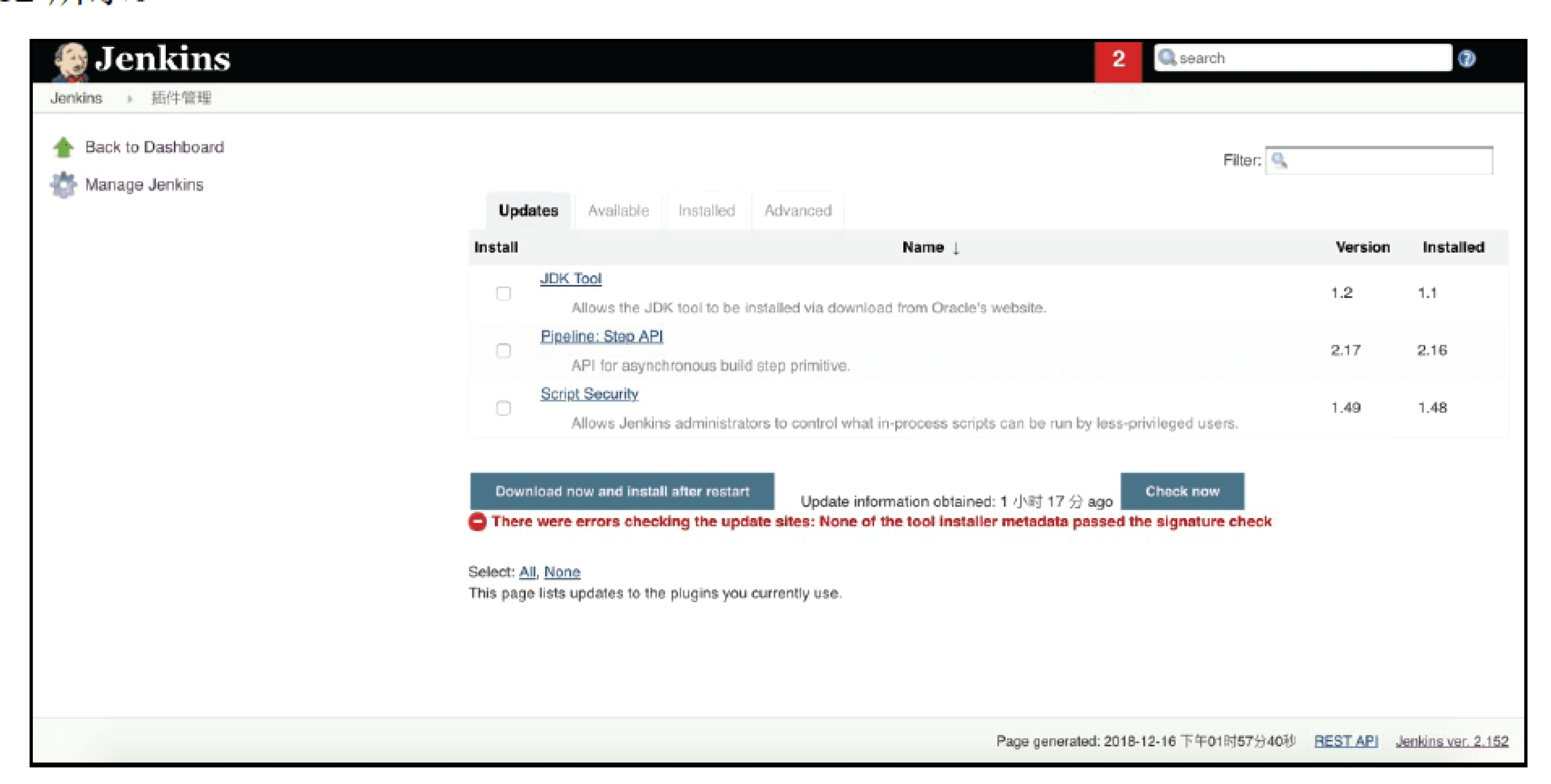


图 12-12　Jenkins 插件页面

3. 全局工具配置

在全局工具配置处配置服务器或者物理机的 JDK、Git 和 Maven 环境变量，如图 12-13 所示。

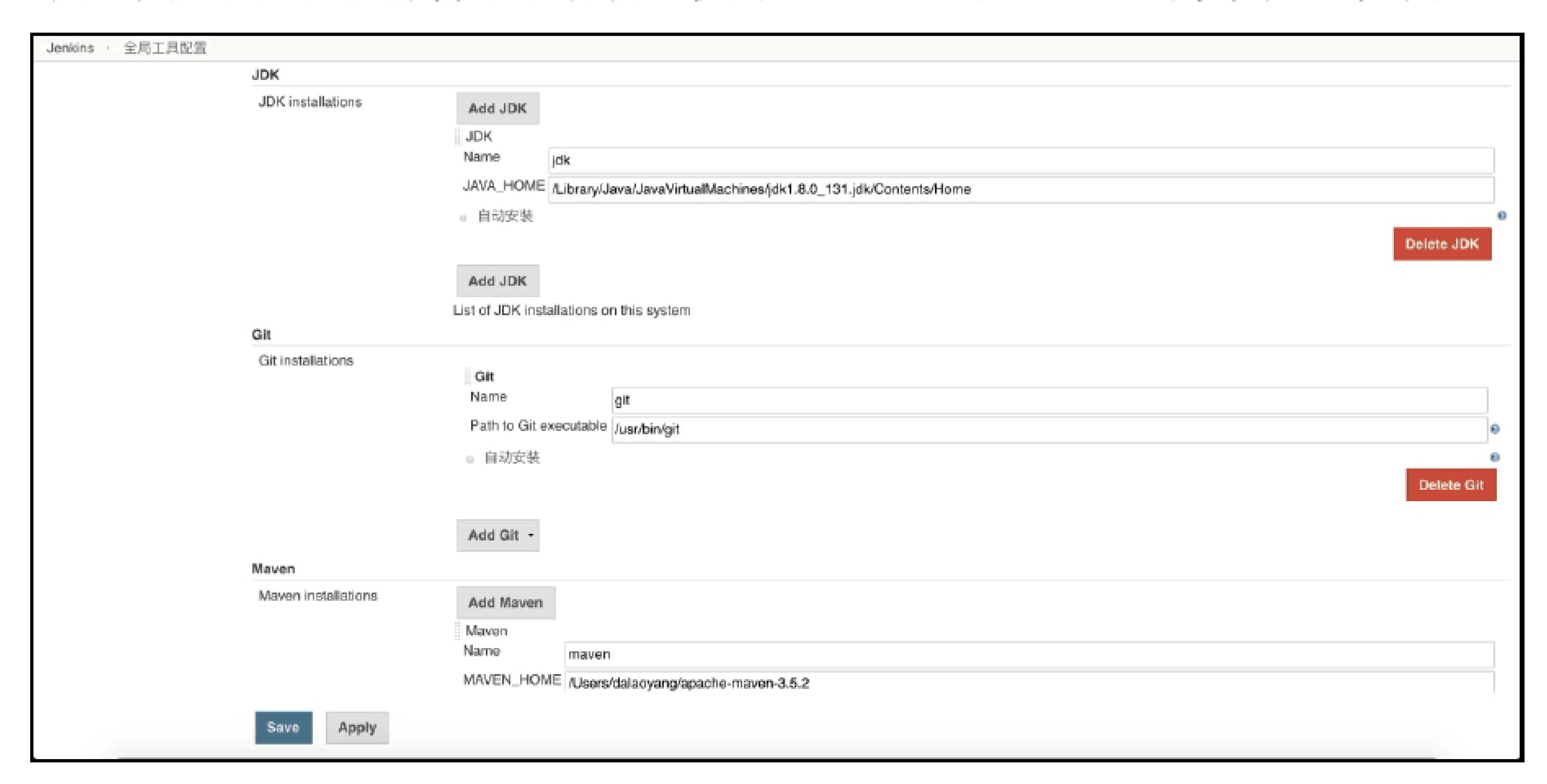


图 12-13　Jenkins 全局工具配置页面

4. 构建项目

创建一个自由风格的软件项目即可，然后为应用起一个名称，这里起名为 Spring Boot-Project，如图 12-14 所示。



图 12-14　Jenkins 构件项目页面

然后单击下方的 OK 按钮。其实构建项目很简单，先创建几个参数，选择参数化构建，如图 12-15 和图 12-16 所示。

使用的参数（其实构建参数不是必须要这样做，只是方便后面使用。由于范例项目比较简单，可以略过）说明如下。

- JAR_NAME：Spring Boot 应用程序打成 JAR 包后的文件名称。
- CHECK_URL：测试项目是否启动成功的地址。
- JDK_PATH：JDK 地址，启动 JAR 时使用。

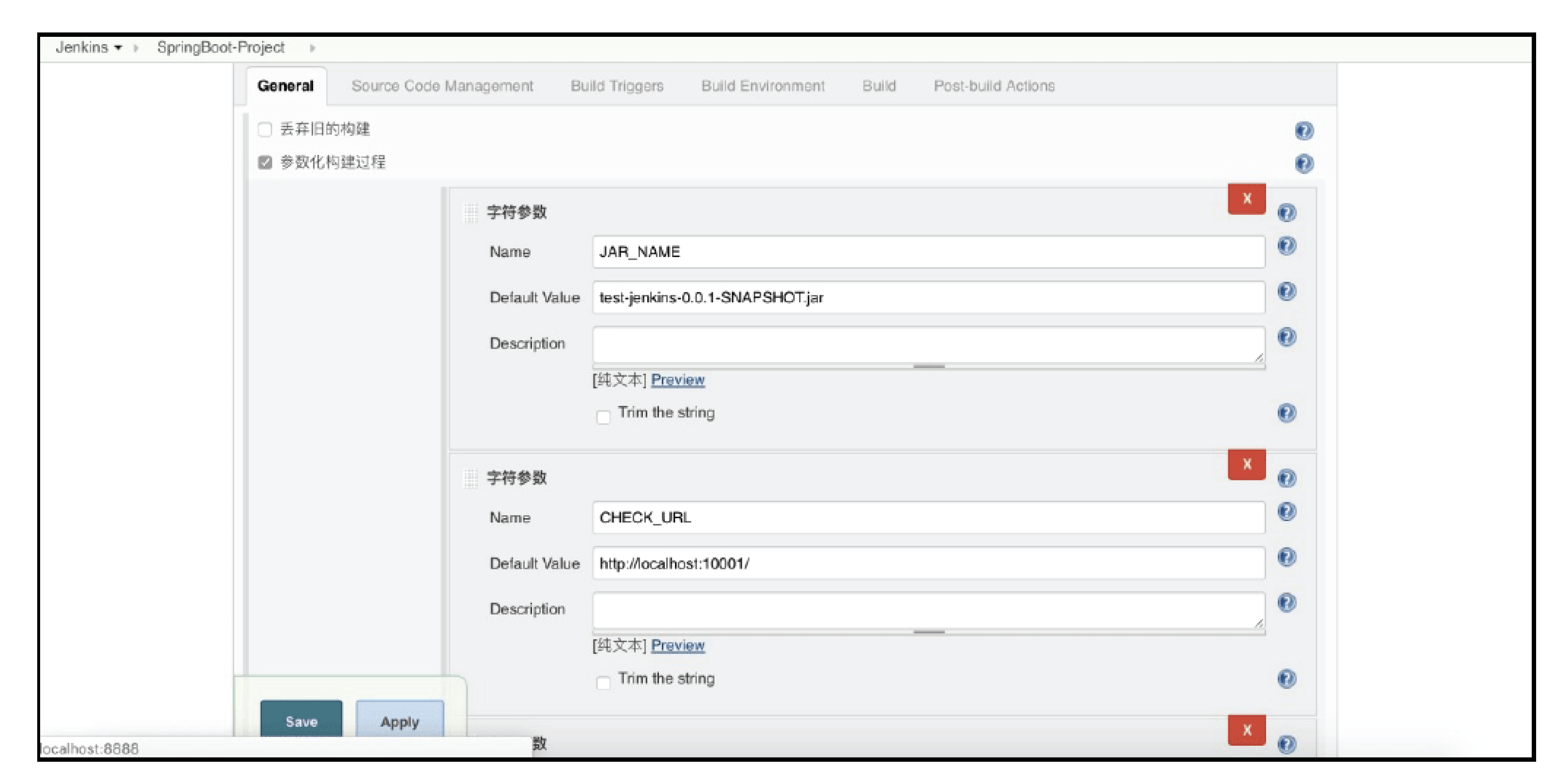


图 12-15　Jenkins 参数化构建页面-1



图 12-16　Jenkins 参数化构建页面-2

接下来配置 Git 应用（本文的 Git 是公开项目，所以无须配置用户名、密码等信息，如果需要校验权限，就需要配置用户名和密码），如图 12-17 所示。

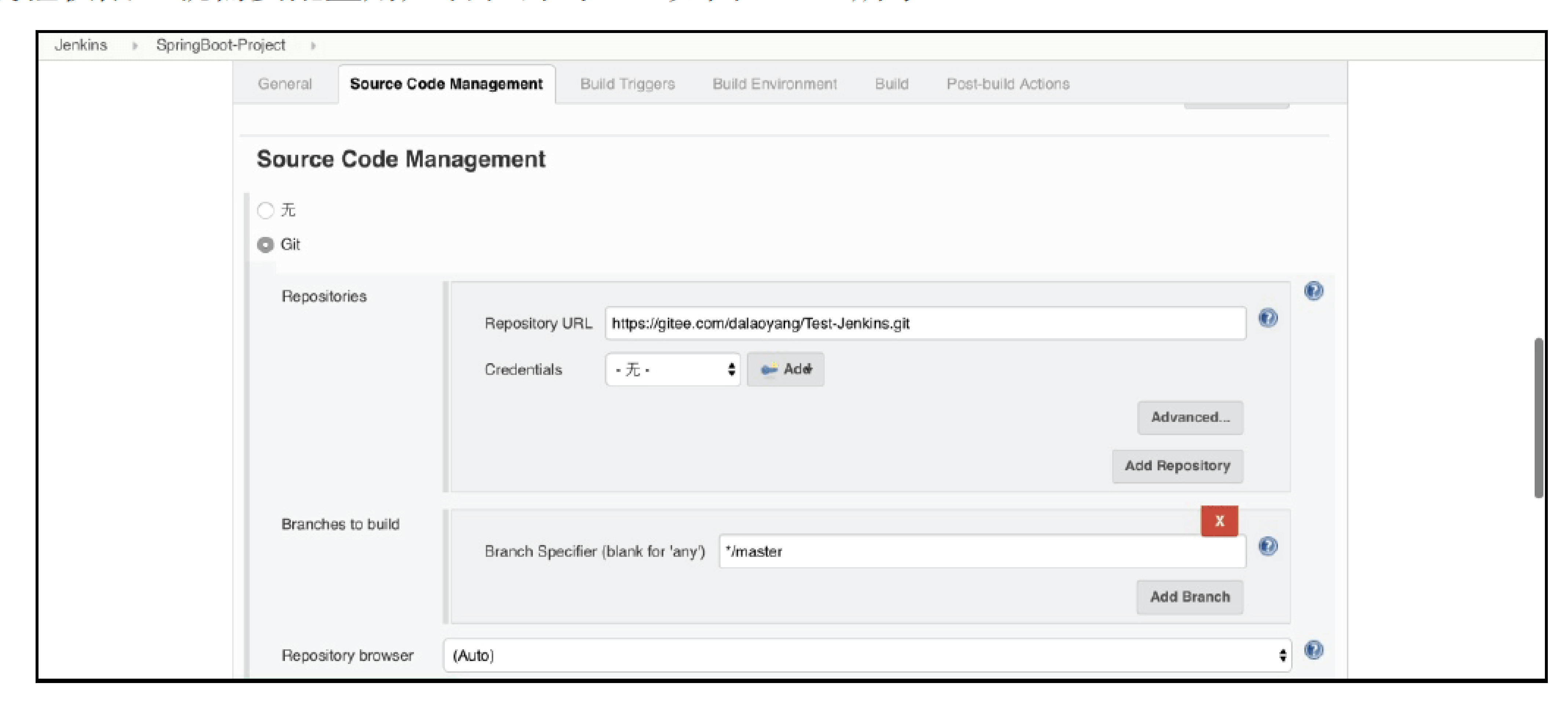


图 12-17　Jenkins-Git 配置

最后，我们只需要在 Build 部分配置需要执行的命令即可，如代码清单 12-10 所示。

代码清单 12-10　Build 执行脚本

```
#程序打包，跳过 test 文件
mvn clean install -Dmaven.test.skip=true;
#执行 shell 启动应用程序
/Users/dalaoyang/Downloads/startup.sh $JAR_NAME $CHECK_URL $WORKSPACE
$JDK_PATH
```

上面的命令中，我们通过对应用打包，然后执行了一个脚本并且传入了 4 个参数，其中 startup.sh 如代码 12-11 清单所示。

代码清单 12-11　Build 执行脚本

```
#保证 Jenkins 进程不被杀掉
export BUILD_ID=dontKillMe

JAR_NAME=${1}
CHECK_URL=${2}
WORKSPACE=${3}
JDK_PATH=${4}

if  [ ! -n "${JAR_NAME}" ] ;then
    echo "参数 1. JAR_NAME 为空"
    exit 1
fi

if  [ ! -n "${CHECK_URL}" ] ;then
    echo "参数 2. CHECK_URL 为空"
    exit 1
fi
if  [ ! -n "${WORKSPACE}" ] ;then
    echo "参数 3. WORKSPACE 为空"
    exit 1
fi
if  [ ! -n "${JDK_PATH}" ] ;then
    echo "参数 4. JDK_PATH 为空"
    exit 1
fi

PID=`ps -ef |grep ${JAR_NAME} |grep -v grep | awk '{print $2}'`
if [ ! "$PID" ];then # 这里判断当前 JAR 进程是否存在
    echo "进程不存在"
else
```

```
    echo "进程存在 杀死进程 PID$PID"
    kill -9 $PID
  fi

  #后台启动 jar
  echo "服务启动中"
  nohup ${JDK_PATH} -jar ${WORKSPACE}/target/${JAR_NAME} &


  #服务检查
  CHECK_ATTEMPTS=20
  CHECK_TIMEOUT=6


  #服务启动检测
  ONLINE=false
  echo "检测服务启动状态"
  for (( i=1; i<=${CHECK_ATTEMPTS}; i++ ))
  do
    CODE=`curl -sL --connect-timeout 20 --max-time 30 -w "%{http_code}\\n"
"${CHECK_URL}" -o /dev/null`
    echo "服务检测返回结果：$CODE"
    if [ "${CODE}" = "200" ]; then
      echo "已检测到服务：${CHECK_URL}"
      sleep 10
      ONLINE=true
      break
    else
      echo "未检测到服务，等待 ${CHECK_TIMEOUT} 秒后重试"
      sleep ${CHECK_TIMEOUT}
    fi
  done
  if $ONLINE; then
    echo "服务检查结束，服务启动正常"
    exit 0
  else
    echo "服务检查结束，服务启动失败"
    exit 1
  fi
```

解释一下脚本内容，大致分为如下几步：

（1）在 Jenkins 默认配置下，执行完成后会杀死所有子进程，所以即使是使用 nohup 启动的 Java 程序，也会被关闭，所以在文件开始就指定当前进程不被关闭。

（2）环境参数的校验，因为参数都是需要使用的，所以这里的校验不能为空。

（3）判断当前 JAR 是否已经被启动，如果已经启动，再次启动就会无法启动，所以先检查是否有当前 JAR 文件的端口，如果有，就先关闭。

（4）使用 nohup 后台启动 JAR。

（5）判断服务是否成功启动。这里是通过请求应用程序的一个路径，如果请求成功，就认为启动正常，否则启动失败。

5. 启动应用程序

单击 Build with Parameter 按钮，如图 12-18 所示，然后单击下方的 Build 按钮。

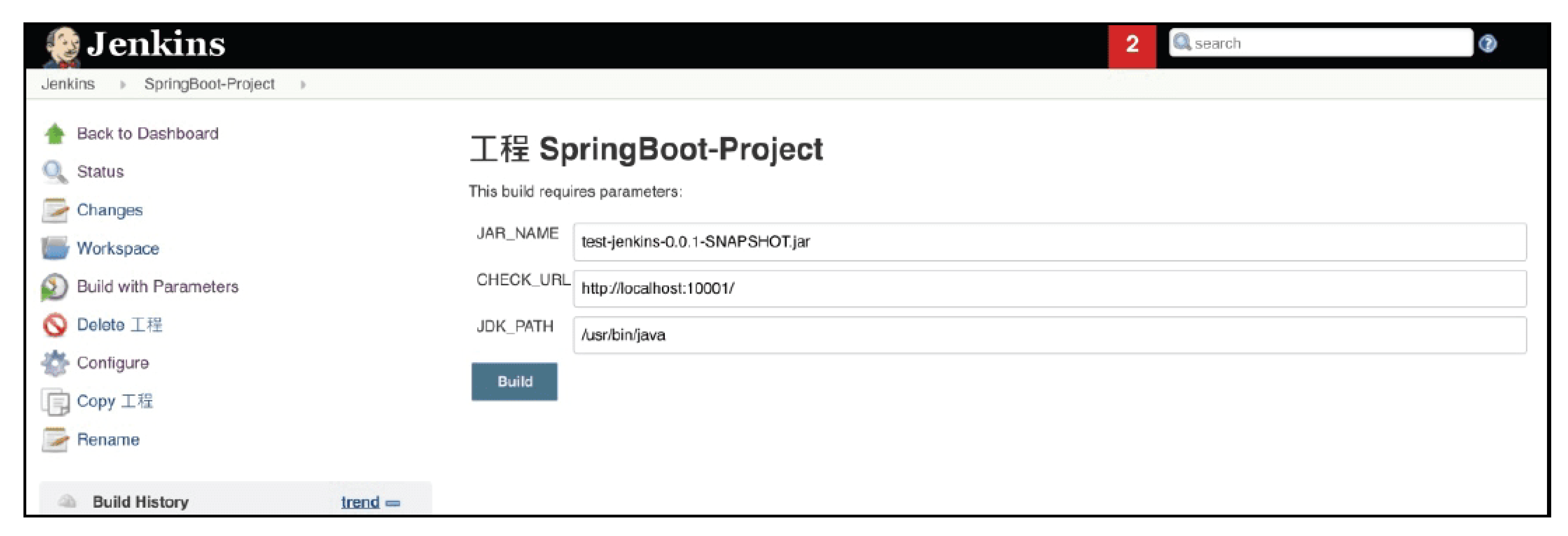


图 12-18　Jenkins-启动应用程序页面

然后查看控制台，可以看到如图 12-19 所示的内容。

```
initialized with port(s): 10001 (http)
2018-12-16 13:43:47.080  INFO 6929 --- [           main] o.apache.catalina.core.StandardService   : Starting
service [Tomcat]
2018-12-16 13:43:47.080  INFO 6929 --- [           main] org.apache.catalina.core.StandardEngine  : Starting
Servlet Engine: Apache Tomcat/9.0.13
2018-12-16 13:43:47.112  INFO 6929 --- [           main] o.a.catalina.core.AprLifecycleListener   : The APR based
Apache Tomcat Native library which allows optimal performance in production environments was not found on the
java.library.path:
[/Users/dalaoyang/Library/Java/Extensions:/Library/Java/Extensions:/Network/Library/Java/Extensions:/System/Librar
y/Java/Extensions:/usr/lib/java:.]
2018-12-16 13:43:47.254  INFO 6929 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing
Spring embedded WebApplicationContext
2018-12-16 13:43:47.255  INFO 6929 --- [           main] o.s.web.context.ContextLoader            : Root
WebApplicationContext: initialization completed in 2350 ms
2018-12-16 13:43:47.608  INFO 6929 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing
ExecutorService 'applicationTaskExecutor'
2018-12-16 13:43:47.999  INFO 6929 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started
on port(s): 10001 (http) with context path ''
2018-12-16 13:43:48.006  INFO 6929 --- [           main] com.dalaoyang.TestJenkinsApplication     : Started
TestJenkinsApplication in 3.894 seconds (JVM running for 4.599)
2018-12-16 13:43:49.614  INFO 6929 --- [io-10001-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing
Spring DispatcherServlet 'dispatcherServlet'
2018-12-16 13:43:49.614  INFO 6929 --- [io-10001-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing
Servlet 'dispatcherServlet'
2018-12-16 13:43:49.623  INFO 6929 --- [io-10001-exec-1] o.s.web.servlet.DispatcherServlet        : Completed
initialization in 9 ms
服务检测返回结果：200
已检测到服务：http://localhost:10001/
服务检查结束，服务启动正常
Process leaked file descriptors. See https://jenkins.io/redirect/troubleshooting/process-leaked-file-descriptors
for more information
Finished: SUCCESS
```

图 12-19 Jenkins-启动控制台页面

在浏览器访问 http://localhost:10001，可以看到控制台有如下输出：

```
SpringBoot-Jenkins-Project
```

Jenkins 可以做的还有很多，案例中只是通过 Jenkins 拉取代码在本地执行，其实还可以发布到远程服务器，在 Jenkins 配置好远程服务器，然后将 JAR 或者 WAR 文件及 Shell 脚本发送过去执行即可。由于篇幅原因，这里不再赘述。

12.7 小 结

本章对 Spring Boot 项目的运行和部署进行了学习，并且介绍了如何使用 Jenkins 自动化部署 Spring Boot 应用。相信经过本章的学习，读者对 Spring Boot 应用程序的部署已经游刃有余，可以在实际工作中部署时得心应手。

第 13 章

Spring Boot 实战之博客系统

前面的章节对 Spring Boot 进行了阶段性的学习，本章将进行实战演练，利用 Spring Boot 框架制作一个博客系统。

13.1 博客的制作思路

很多开发者都喜欢利用一些平台进行技术分享，如 CSDN、简书、掘金等。当然，也有很多开发者喜欢制作属于自己的博客进行技术分享，如今比较常用的开源博客有 Hexo 和 WordPress。虽然这些开源博客都很不错，但是作为开发者，开发一个个人博客也是很有意思的事情。本章将带领大家开发一个属于自己的博客系统。

制作博客的思路分为如下几步：

（1）静态模板项目制作，将 HTML 静态项目改为 Thymeleaf 项目，使用 Controller 进行跳转。

（2）实体设计，因为使用的是 Spring Data JPA，实体设计会决定数据库表的结构。

（3）后台方法代码编写，包含查询数据库、封装数据等。

（4）渲染数据，将后台查询出来的数据动态渲染到 Thymeleaf。

13.1.1 博客布局介绍

以从网上下载的静态博客模板为例，首先来看博客的布局设计。博客的首页分为上、中、下 3 部分，其中上方和下方是本博客的公共部分，中间则为每个导航动态显示的内容，分别说明如下。

（1）上方

左边是博客的 LOGO，这里以笔者的网名为例，当然读者也可以根据自己的网名或喜欢的名

称给博客命名，单击博客 LOGO 返回博客首页。右边是文章的导航栏，导航栏分为 5 个模块，即首页、博客页、搜索页、关于页和联系页，单击后可跳转至对应模块。导航栏如图 13-1 所示。

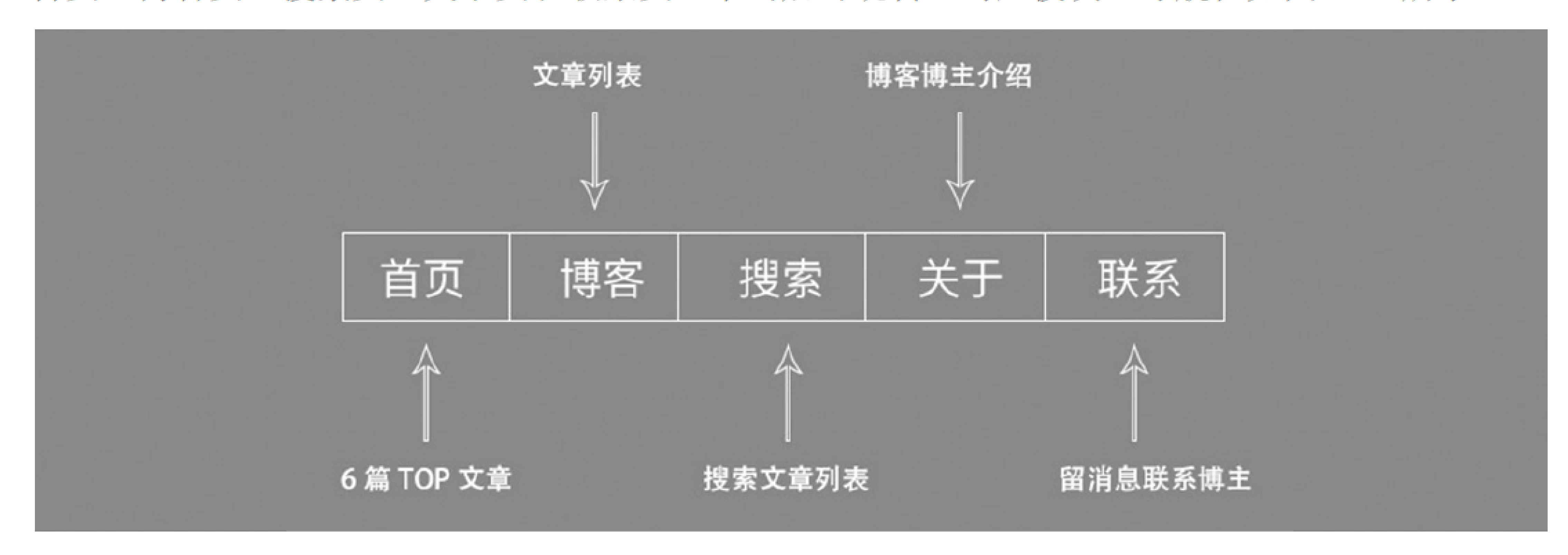


图 13-1　导航栏

（2）中间

内容为各个模块或功能显示的内容，稍后会详细介绍。

（3）下方

下方分为三部分，左边是标签列表，单击标签列表中的标签会进入标签列表页，单击标签列表页中的某个标签会进入当前标签对应的文章列表页。中间为友情链接列表页，单击链接名会跳转至对应的地址。右边为网站的简介，这里只配置了博客的域名和备案号，读者可以根据自己的需求进行配置。

13.1.2　博客功能介绍

博客功能其实就是页面中间部分显示的内容，分为以下几个功能。

（1）首页：首页显示的内容为博客中最后置顶的 6 篇文章。

（2）博客页：显示博客中的 10 篇文章，显示内容包含文章标题、文章作者、头像、文章简介，本页包含分页。

（3）搜索页：显示博客中根据关键字搜索出的 10 篇文章，显示内容包含文章标题、文章作者、头像、文章简介，本页包含分页。

（4）关于页：关于博主的介绍，这里可以根据系统配置设置一篇文章为关于页的内容。

（5）联系页：可以发送一条消息留言给博主。

（6）标签列表页：显示全部标签，无分页。

（7）标签对应博客页：根据选择的标签名称显示对应博客列表，暂无分页，读者需要可以加分页。

（8）文章详情页：显示文章的详细内容。

13.2 博客模板制作

本章博客系统使用模板框架 Thymeleaf。接下来简单介绍如何将静态 HTML 页面项目修改为 Thymeleaf 静态项目。

这里需要提取 4 个公共模块，提取公共模块与提取公共方法的原因一致，为了避免项目内有太多冗余代码，方便后期修改。这里的公共模块分别说明如下。

- 公共头模块：主要用于引入 CSS 资源和头部信息。
- 公共导航模块：主要用于引入导航。
- 公共底部模块：主要用于底部信息。
- 公共 Js 模块：主要用于引入 JS 资源。

这里以首页为例，其他页面的方法类似。首先制作一个公共头模块，新建 HTML 页面，这里设置名称为 common_head.html，将对应资源放入，使用 th:fragment 标明当前模块的名称，如代码清单 13-1 所示。

代码清单 13-1　博客实战项目-公共头模块内容

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th=
"http://www.thymeleaf.org">
<!DOCTYPE html>
<html lang="en">
<head th:fragment="commonHeader">
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="">
    <meta name="author" content="">
    <title>这是一个模板博客</title>
    <link rel="stylesheet" th:href="@{/css/bootstrap.css}" type="text/css">
    <link rel="stylesheet" th:href="@{/css/main.css}" type="text/css">
    <link rel="stylesheet" th:href="@{/css/page.css}">
    <link rel="stylesheet" th:href="@{/css/search.css}">
    <link rel="stylesheet"  th:href="@{/font-awesome-4.4.0/css/
font-awesome.min.css}">

</head>
</html>
```

接下来制作其他 3 个公共模块，新建原理与公共头模块一致。公共导航模块如代码清单 13-2 所示。

代码清单 13-2 博客实战项目-公共导航模块内容

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">
<div th:fragment="commonNavigation">
    <div class="navbar navbar-inverse navbar-static-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle"
data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" th:href="@{index}">DALAOYANG</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav navbar-right">
                    <li><a th:href="@{/index}">首页</a></li>
                    <li><a th:href="@{/blog}">博客</a></li>
                    <li><a th:href="@{/search}">搜索</a></li>
                    <li><a th:href="@{/about}">关于</a></li>
                    <li><a th:href="@{/contact}">联系</a></li>
                </ul>
            </div>
        </div>
    </div>
</div>
```

公共底部模块如代码清单 13-3 所示。

代码清单 13-3 博客实战项目-公共底部模块内容

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">
<div th:fragment="commonFooter">
    <div id="footer">
        <div class="container">
            <div class="row">
                <div class="col-lg-4">
                    <h4><a th:href="@{/tag}">标签</a></h4>
                    <p class="footer-tags">
                        <a>SpringBoot</a>
                        <a>SpringCloud</a>
                        <a>Nginx</a>
```

```
                        <a>Linux</a>
                        <a>Mybatis</a>
                        <a>Spring Data Jpa</a>
                        <a>JDBC</a>
                        <a>Eureka</a>
                    </p>

                </div>

                <div class="col-lg-4">
                    <h4>友情链接</h4>
                    <p>
                        <a href="https://blog.csdn.net/qq_33257527">
CSDN</a><br/>
                        <a href="https://www.jianshu.com/u/128b6effde53">简书
</a><br/>
                        <a href="https://www.imooc.com/u/6841077">慕课网</a>
                    </p>
                </div>

                <div class="col-lg-4">
                    <h4>关于网站</h4>
                    <p>Copyright © 2018. Dalaoyang.cn  </p>
                    <p>All rights reserved.</p>
                    <p>辽 ICP 备 17014944 号-1</p>

                </div>

            </div>

        </div>
    </div>
</div>
```

公共 JS 模块如代码清单 13-4 所示。

代码清单 13-4　博客实战项目-公共 JS 模块内容

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">

<!DOCTYPE html>
<html lang="en">
<body>
<div th:fragment="onLoadJs">
    <!-- jQuery -->
    <script th:src="@{/js/jquery-1.10.2.min.js}"></script>
```

```
        <script type="text/javascript" th:src="@{/js/bootstrap.min.js}">
</script>
        <!-- Custom Theme JavaScript -->
        <script th:src="@{/js/page.js}"></script>
        <script th:src="@{/js/hover.zoom.js}"></script>
        <script th:src="@{/js/hover.zoom.conf.js}"></script>
        <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media
queries -->
        <!--[if lt IE 9]>
        <script th:src="@{/js/html5shiv.js}"></script>
        <script th:src="@{/js/respond.min.js}"></script>
        <![endif]-->
    </div>
    </body>
    </html>
```

接下来看一下首页的代码。使用 th:include 引入对应资源，方式为 th:include="公共模块位置::公共模块名称"，完整内容如代码清单 13-5 所示。

代码清单 13-5　博客实战项目-首页内容

```
    <html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">
    <html lang="en">

     <head th:include="common/common_head::commonHeader"></head>

     <body>

     <div th:include="common/common_navigation :: commonNavigation"></div>
       <div class="container pt">
           <div class="row mt centered">
               <div class="col-lg-4">
                   <a class="zoom green" th:href="@{/articleDetailDemo}"><img
class="img-responsive" th:src="@{/img/portfolio/port01.jpg}" alt="" /></a>
                   <p>SpringBoot</p>
               </div>
               <div class="col-lg-4">
                   <a class="zoom green" th:href="@{/articleDetailDemo}"><img
class="img-responsive" th:src="@{/img/portfolio/port02.jpg}" alt="" /></a>
                   <p>SpringCloud</p>
               </div>
               <div class="col-lg-4">
                   <a class="zoom green" th:href="@{/articleDetailDemo}"><img
class="img-responsive" th:src="@{/img/portfolio/port03.jpg}" alt="" /></a>
                   <p>Linux</p>
```

```
            </div>
        </div>
        <div class="row mt centered">
            <div class="col-lg-4">
                <a class="zoom green" th:href="@{/articleDetailDemo}"><img
class="img-responsive" th:src="@{/img/portfolio/port04.jpg}" alt="" /></a>
                <p>微服务</p>
            </div>
            <div class="col-lg-4">
                <a class="zoom green" th:href="@{/articleDetailDemo}"><img
class="img-responsive" th:src="@{/img/portfolio/port05.jpg}" alt="" /></a>
                <p>多线程</p>
            </div>
            <div class="col-lg-4">
                <a class="zoom green" th:href="@{/articleDetailDemo}"><img
class="img-responsive" th:src="@{/img/portfolio/port06.jpg}" alt="" /></a>
                <p>消息队列</p>
            </div>
        </div>
    </div>
  <div th:include="common/common_footer :: commonFooter"></div>
  </body>
  <div th:include="common/common_onload_js :: onLoadJs"></div>
</html>
```

其他页面的制作原理类似，这里就不多说了。如果读者感兴趣，可以自己尝试制作，也可以在笔者提供的源代码的基础上开发。

关于跳转都是使用 Controller，目前的数据都是静态数据，比如首页的跳转如代码清单 13-6 所示。

代码清单 13-6　博客实战项目-IndexController 内容

```
@Controller
public class IndexController {

    @GetMapping(value = {"/","index"})
    public String index(){
        return "index";
    }
}
```

13.3 效果展示

截至目前，已经完成了 Thymeleaf 模板项目的制作。接下来我们来看页面效果。

首页如图 13-2 所示。

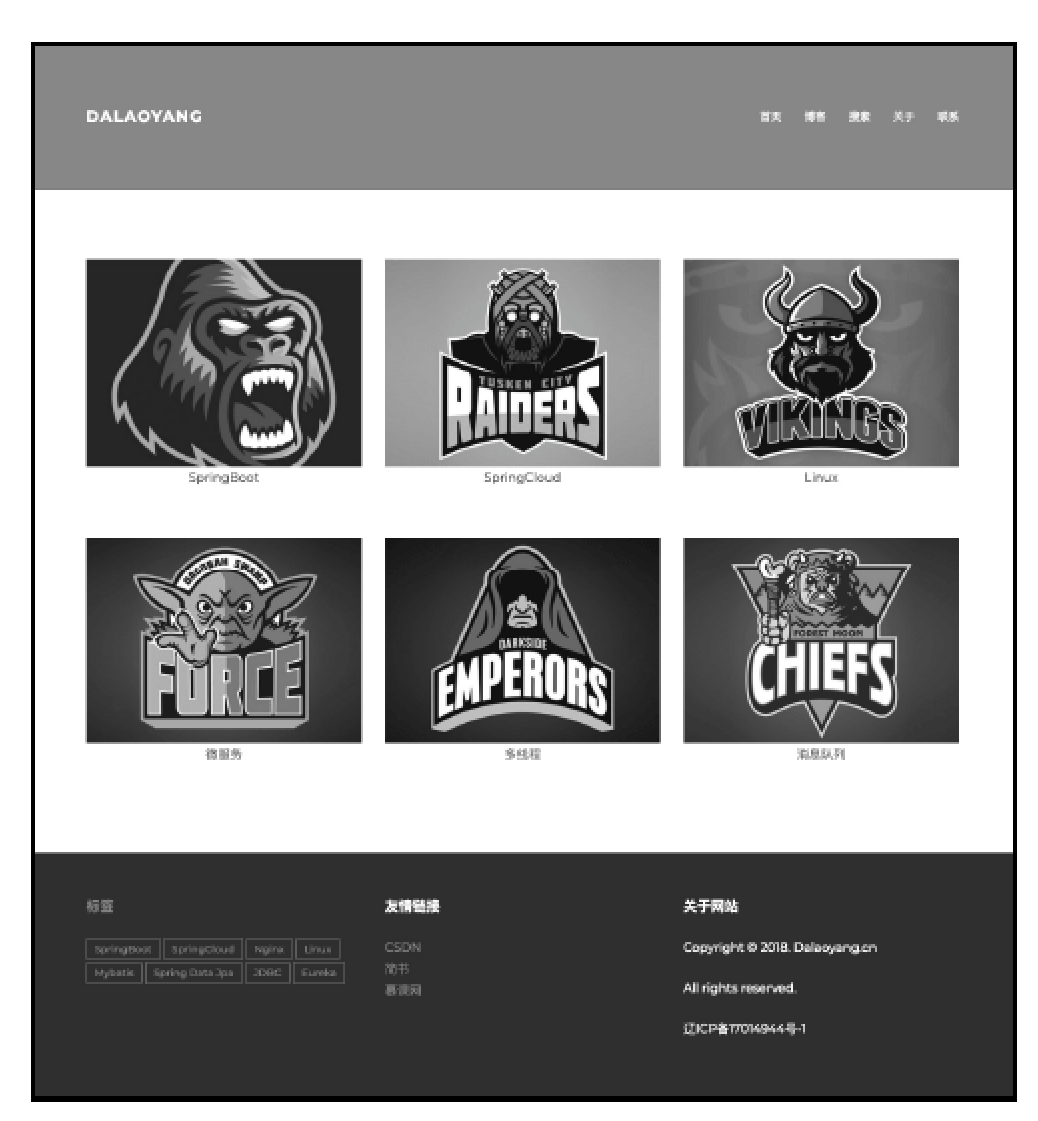


图 13-2　首页展示图

博客页如图 13-3 所示。



图 13-3　博客页展示图

搜索页如图 13-4 所示。

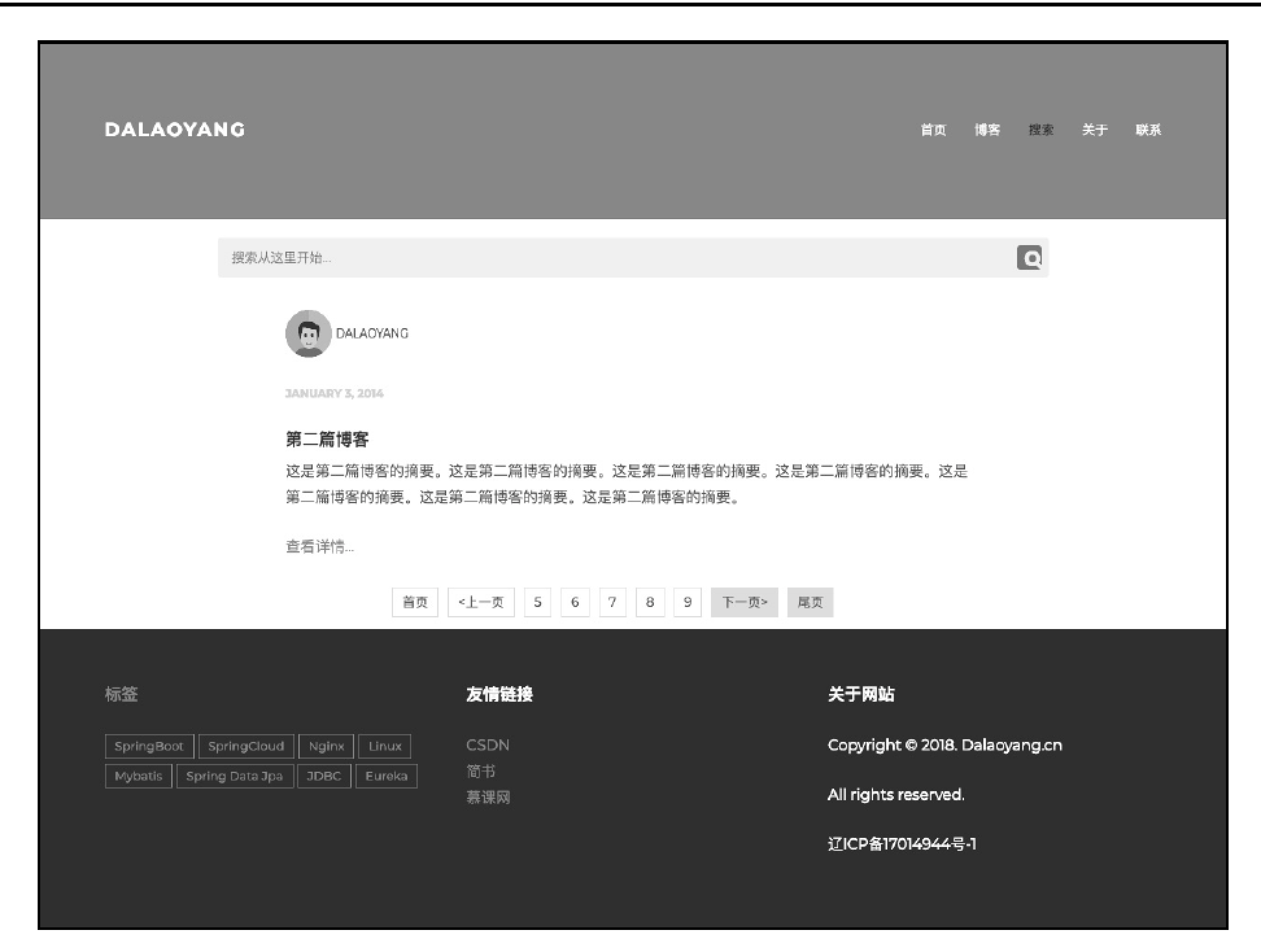


图 13-4 搜索页展示图

关于页如图 13-5 所示。

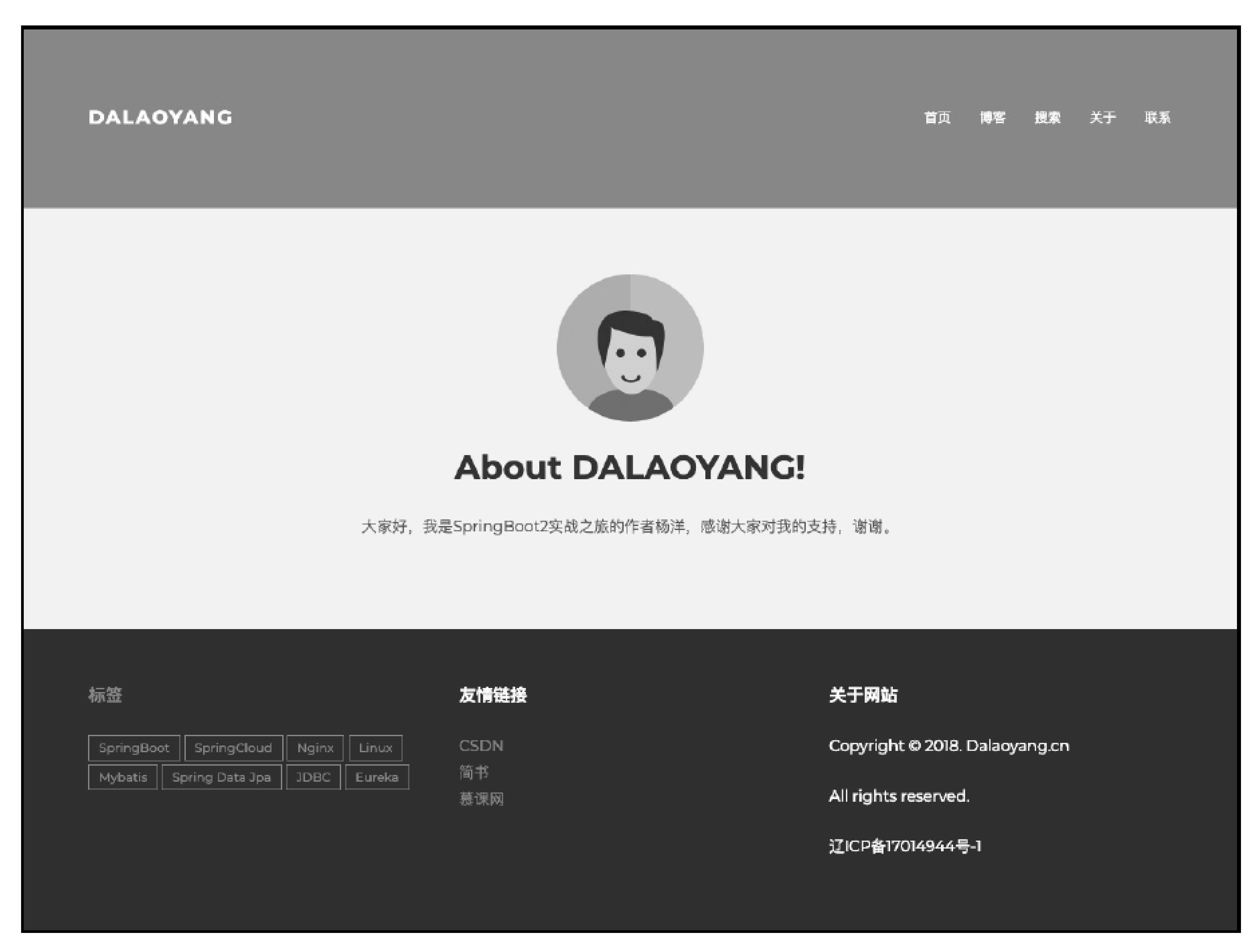


图 13-5 关于页展示图

联系页如图 13-6 所示。

图 13-6　联系页展示图

标签页如图 13-7 所示。

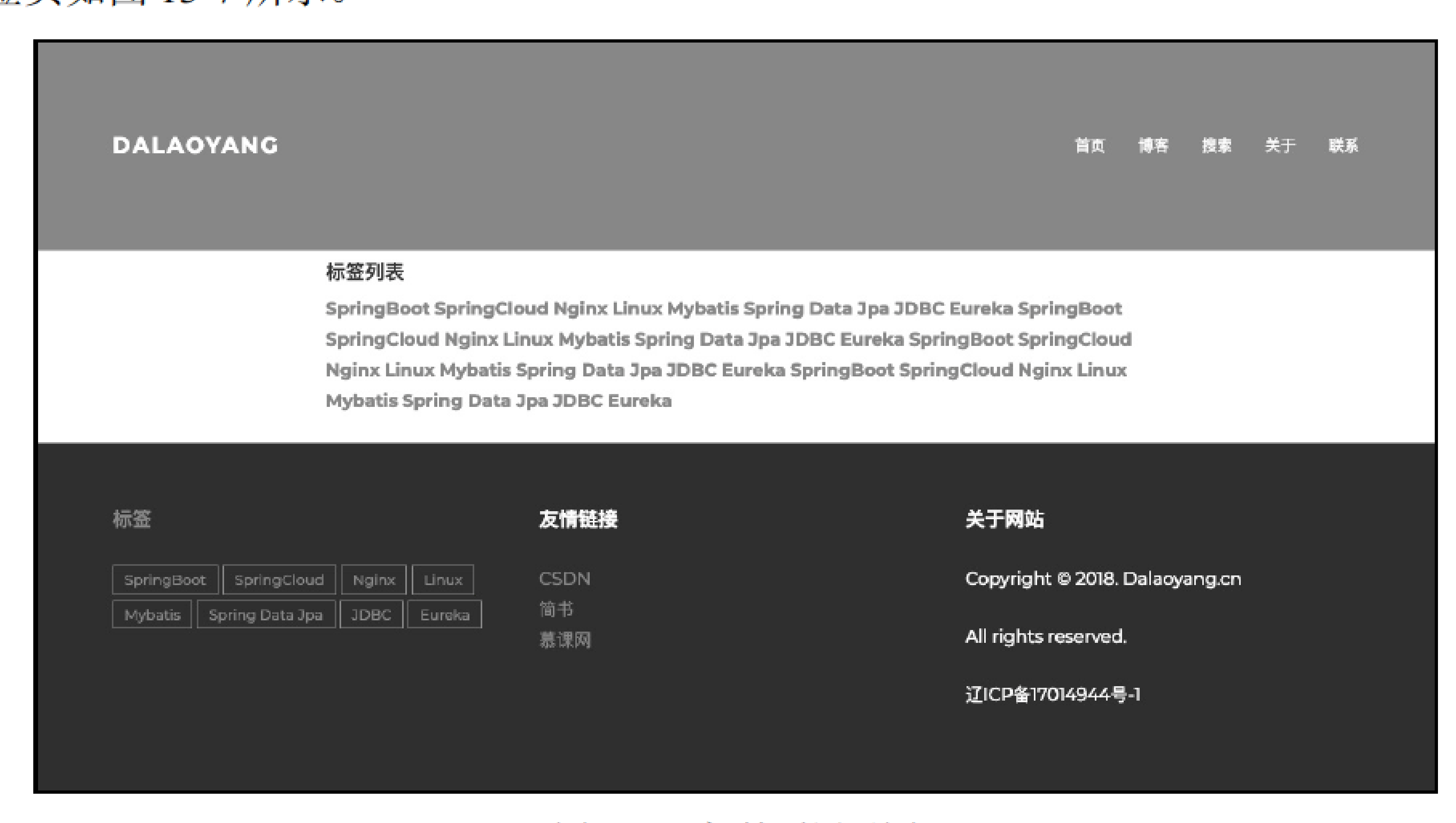


图 13-7　标签页展示图

标签对应博客页内容与博客列表页一致，这里就不展示了。文章详情页如图 13-8 所示。



图 13-8　文章详情页展示图

13.4　依赖配置

正如前面介绍的，本文使用的前端模板框架是 Thymeleaf，数据库为 MySQL 数据库。因为只有一些简单的查询，所以 ORM 层使用 Spring Data JPA，并且加入了 pegdown 依赖支持 Markdown 格式文章转换，依赖内容如代码清单 13-7 所示。

代码清单 13-7　博客项目-依赖配置

```
<dependencies>
      <!-- jpa-->
      <dependency>
          <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <!-- thymeleaf-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <!-- web-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- mysql-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <!-- 去除 thymeleaf 严格校验-->
        <dependency>
            <groupId>net.sourceforge.nekohtml</groupId>
            <artifactId>nekohtml</artifactId>
            <version>1.9.22</version>
        </dependency>
        <!-- lombok-->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.16.20</version>
        </dependency>
        <!-- 转换 markdown-->
        <dependency>
            <groupId>org.pegdown</groupId>
            <artifactId>pegdown</artifactId>
            <version>1.6.0</version>
        </dependency>
    </dependencies>
```

13.5 配 置 文 件

配置文件配置类 Thymeleaf 的缓存设置，这里暂时设置为 false，当全部开发完成后，可以设置为 true，剩余只配置了数据库和 JPA，端口号设置的是 10000。完整配置如代码清单 13-8 所示。

代码清单 13-8　博客项目-配置文件内容

```
##端口号
server.port=10000
##禁用 thymeleaf 缓存
spring.thymeleaf.cache=false

##数据库配置
##数据库地址
spring.datasource.url=jdbc:mysql://localhost:3306/springbootBlog?characte
rEncoding=utf8&useSSL=false
##数据库用户名
spring.datasource.username=root
##数据库密码
spring.datasource.password=root
##数据库驱动
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

##none　启动时不做任何操作
spring.jpa.hibernate.ddl-auto=update
##控制台打印 sql
spring.jpa.show-sql=true
```

13.6　后台实体

接下来介绍博客需要的几张表，由于使用 JPA 进行操作，因此创建了对应的实体类，可以在数据库中生成对应的表。

13.6.1　文章表

文章表主要记录文章信息，是博客最主要的表，表名为 article，针对当前案例设置了以下几个字段属性。

- articleId：文章主键 ID，设置主键自增列。
- articleName：文章名称。
- articleContent：文章内容，设置字段类型为 TEXT。
- articleAuthors：文章作者。
- articleInputDate：文章录入日期，设置字段类型为 Date。
- articleReadingTime：文章阅读次数。
- isTop：是否置顶。

- isEnable：是否启用（可以理解为是否发布）。
- tagList：设置与标签表的多对多关系。

接下来介绍的几个属性是在项目内使用的，并非数据库字段。

- imageNo：文章对应图片，案例中只在首页置顶文章中设置了对应的图片。
- articleIntroduction：文章简介，用于文章列表页、搜索页、标签文章页展示文章简介，案例中是将文章内容去除 HTML 标签内容后，截取 100 个字符组成。
- articleShowContent：展示文章内容，因为支持 Markdown 语法，所以需要将 Markdown 格式文章内容直接存储到数据库中，案例中使用这个字段展示文章详细内容。

完整 article 实体类内容如代码清单 13-9 所示。

代码清单 13-9 博客项目-article 实体类内容

```
@Entity
@Table(name = "article")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Article implements Serializable {

    private static final long serialVersionUID = 4967006908141911451L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long articleId;
    private String articleName;
    @Lob
    @Column(columnDefinition = "TEXT")
    private String articleContent;
    private String articleAuthors;
    @Temporal(TemporalType.DATE)
    private Date articleInputDate;
    private Integer articleReadingTime;
    private Integer isTop;
    private Integer isEnable;

    @ManyToMany
    @JoinTable(name = "articleTag", joinColumns = {@JoinColumn(name =
"articleId")}, inverseJoinColumns = {@JoinColumn(name = "tagId")})
    private List<Tag> tagList;

    //项目内使用，非数据库字段
    @Transient
    private int imageNo;
```

```
    @Transient
    private String articleIntroduction;
    @Transient
    private String articleShowContent;

}
```

13.6.2 标签表

标签表主要记录标签信息，表名为 tag，针对当前案例设置如下字段。

- tagId：标签表主键 ID，设置主键自增列。
- tagName：标签名称。
- tagInputDate：标签录入日期，设置字段类型为 Date。
- articleList：设置与文章表的多对多关系。

完整 tag 实体类内容如代码清单 13-10 所示。

代码清单 13-10 博客项目-tag 实体类内容

```
@Entity
@Table(name = "tag")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Tag implements Serializable {

    private static final long serialVersionUID = -7536613142331362542L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long tagId;
    private String tagName;
    @Temporal(TemporalType.DATE)
    private Date tagInputDate;

    @ManyToMany
    @JoinTable(name = "articleTag", joinColumns = {@JoinColumn(name =
"tagId")}, inverseJoinColumns = {@JoinColumn(name = "articleId")})
    private List<Article> articleList;

}
```

13.6.3　链接表

链接表其实可以理解为博客内友情链接存储的数据表，表名为 link，针对当前案例设置如下字段。

- linkId：链接表主键 ID，设置主键自增列。
- linkName：友情链接名称。
- linkUrl：友情链接地址。
- remark：友情链接备注。

完整 link 实体类内容如代码清单 13-11 所示。

代码清单 13-11　博客项目-link 实体类内容

```
@Entity
@Table(name = "link")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Link implements Serializable {
    private static final long serialVersionUID = -4725937550197599617L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long linkId;
    private String linkName;
    private String linkUrl;
    private String remark;
}
```

13.6.4　消息表

消息表主要用于记录访客在联系页给博主留言的信息，表名为 message，针对当前案例设置如下字段。

- messageId：消息表主键 ID，设置主键自增列。
- email：发消息人的邮箱地址。
- name：发消息人的名称。
- subject：消息的主题。
- messageContent：消息的内容，设置字段类型为 TEXT。
- messageInputDate：消息发送的时间。
- isRead：是否已读。

完整 message 实体类内容如代码清单 13-12 所示。

代码清单 13-12　博客项目-message 实体类内容

```
@Entity
@Table(name = "message")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Message implements Serializable {
    private static final long serialVersionUID = -5529232129767452275L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long messageId;
    private String email;
    private String name;
    private String subject;
    @Lob
    @Column(columnDefinition="TEXT")
    private String messageContent;
    private Date messageInputDate;
    private Integer isRead;
}
```

13.6.5　博客访问记录表

博客访问记录表主要用于记录以日为单位博客的访问次数，表名为 website_access，针对当前案例设置如下字段。

- id：博客访问记录表主键 ID，设置主键自增列。
- accessDate：访问记录的日期，设置字段类型为 Date。
- accessCount：访问的次数。

完整 WebsiteAccess 实体类内容如代码清单 13-13 所示。

代码清单 13-13　博客项目-WebsiteAccess 实体类内容

```
@Entity
@Table(name = "websiteAccess")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class WebsiteAccess implements Serializable {
    private static final long serialVersionUID = 6948407037095536818L;
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Temporal(TemporalType.DATE)
    private Date accessDate;
    private Integer accessCount;
}
```

13.6.6　博客配置表

博客配置表主要用于记录一些博客的配置信息，表名为 website_config，针对当前案例设置了如下字段。

- Id：博客配置表主键 ID，设置主键自增列。
- blogName：博客名称。
- authorName：博主名称。
- aboutPageArticleId：介绍博主页面的文章 ID。
- recordNumber：备案号。
- domainName：域名。
- emailUsername：博主邮箱地址。

完整 WebsiteConfig 实体类内容如代码清单 13-14 所示。

代码清单 13-14　博客项目-WebsiteConfig 实体类内容

```
@Entity
@Table(name = "websiteConfig")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class WebsiteConfig implements Serializable {
    private static final long serialVersionUID = 7023358255818152969L;
    @Id
    private Long Id;
    private String blogName;
    private String authorName;
    private Long aboutPageArticleId;
    private String recordNumber;
    private String domainName;
    private String emailUsername;
}
```

13.7 主 功 能

首先来看本案例应用程序的目录结构，如图 13-9 所示。其中，每个包对应的功能如下。

- config：配置，案例中用于配置拦截器。
- constants：常量，案例中常量类所在包。
- controller：控制层，案例中控制层大多只用于封装数据和页面跳转。
- entity：实体，在 13.6 节已经介绍了。
- init：初始化，用于初始化应用中的数据。
- interceptors：拦截器，案例中用于更新博客访问次数和初始化底部数据。
- repository：数据操作层，用于 JPA 操作数据库。
- service：业务层，案例中用于调用数据操作层和一些业务逻辑处理。
- timer：定时器，案例中的定时器主要用于在每日凌晨创建博客访问数据表记录。
- util：工具，案例中包含两个工具类：Markdown 转换工具类和去除 HTML 标签工具类。

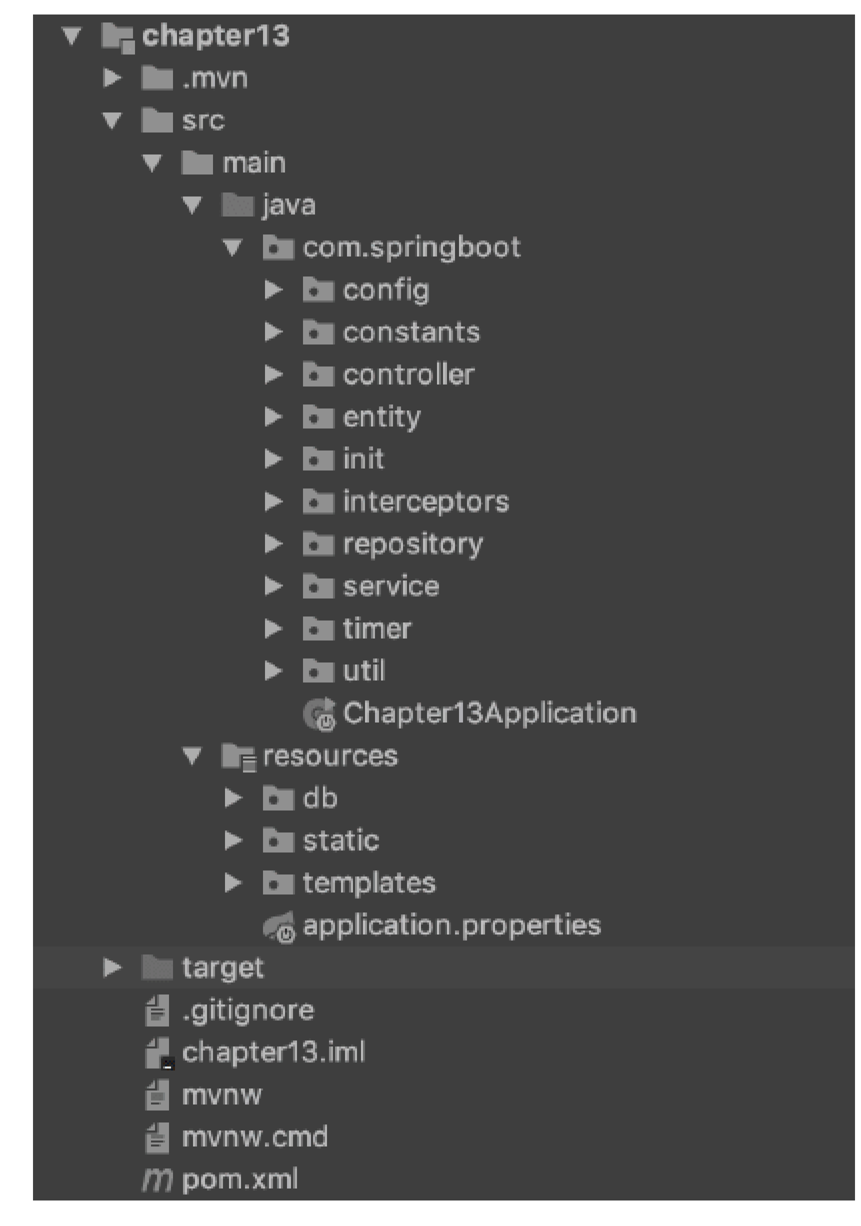


图 13-9 项目目标结构图

头部内容是写死的。当然，读者也可以在配置类中配置，然后动态渲染，底部稍后会介绍。这里以博客页、搜索页、文章详情页和联系页为例介绍博客主流程的代码编写。

13.7.1 博客页

博客页中间部分是由 10 条已发布的文章数据组成的，带有分页，并且根据 articleId 进行倒叙排序，所以我们需要从数据库查询数据展示页面。由于使用了分页并且需要查询已经发布的文章，因此需要传入 Pageable 分页对象和 isEnable 是否发布标示。Repository 层的方法如代码清单 13-15 所示。

代码清单 13-15 博客项目-博客页数据操作层内容

```
Page<Article> findAllByIsEnable(Integer isEnable, Pageable pageable);
```

在 Service 层构建分页所需的 Pageable，然后调用 Repository 层的方法，并且在方法上加入 @Cacheable 注解使用缓存。Service 层内容如代码清单 13-16 所示。

代码清单 13-16　博客项目-博客页 Service 层内容

```
    @Override
    @Cacheable(value = "blogArticle", key = "#page")
    public Page<Article> findBlogArticleList(int page, int size) {
        Pageable pageable = PageRequest.of(page, size, Sort.Direction.DESC,
"articleId");
        return articleRepository.findAllByIsEnable(Constants.YES, pageable);
    }
```

博客页 Controller 层包含两个方法，映射路径分别是/blog 和/blog/{pageNumber}，后者使用路径中的 pageNumber 作为页码进行分页查询，然后将查询到的数据进行循环遍历，文章详情去 HTML 标签设置到 articleIntroduction 属性中。完整 BlogController 内容如代码清单 13-17 所示。

代码清单 13-17　博客项目-博客页 Controller 层内容

```
    @Controller
    public class BlogController {
        @Autowired
        private ArticleService articleService;

        @GetMapping("/blog")
        public String blog(Model model) {
            return this.blog(model, 1);
        }

        @GetMapping("/blog/{pageNumber}")
        public String blog(Model model, @PathVariable Integer pageNumber) {
            if (pageNumber == null) {
                pageNumber = 1;
            }
            Page<Article> articlePage = articleService.findBlogArticleList
((pageNumber-1)*Constants.defaultPageSize,Constants.defaultPageSize );
            List<Article> articleList = articlePage.getContent();
            articleList.forEach(article -> {
                String articleIntroduction = HtmlSpirit.delHTMLTag
(article.getArticleContent());
                article.setArticleIntroduction(articleIntroduction.length() >
100 ? articleIntroduction.substring(0, 100) : articleIntroduction);
            });
            model.addAttribute("articleList", articleList);
            model.addAttribute("totalCount", articlePage.getTotalElements());
            model.addAttribute("pageNumber", pageNumber);
            return "blog";
        }
    }
```

最后查看页面展示，使用 th:each 进行迭代集合，使用 th:text 进行集合内对象内容的展示。这里的内容很简单，就不做太多介绍了。完整内容如代码清单 13-18 所示。

代码清单 13-18 博客项目-博客页前端页面内容

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">
<html lang="en">
<head th:include="common/common_head::commonHeader"></head>
<body>
<div th:include="common/common_navigation :: commonNavigation"></div>

<div>
    <p> </p>
    <div th:each="al,iterStat : ${articleList}">
        <div class="container">
            <div class="row">
                <div class="col-lg-8 col-lg-offset-2">
                    <p><img th:src="@{/img/user.png}" width="50px"
height="50px"> <span th:text="${al.articleAuthors}"></span></p>
                    <p th:text="${al.articleInputDate}"></p>
                    <h4 th:text="${al.articleName}"></h4>
                    <p th:text="${al.articleIntroduction}"></p>
                    <p><a th:href="${'/article/'+al.articleId}">查看详情...
</a></p>
                </div>
            </div>
        </div>
    </div>

<div style="margin-bottom: 50px;">
    <div class="col-md-3" ></div>
    <div class="col-md-6"><ul class="page" maxshowpageitem="5"
pagelistcount="10" id="page"></ul></div>
    <div class="col-md-3"></div>
</div>
</div>
    <div th:include="common/common_footer :: commonFooter"></div>
  </body>
  <div th:include="common/common_onload_js :: onLoadJs"></div>
<script type="text/javascript"  th:inline="javascript">
    var totalCount =  [[${totalCount}]];
    var pageNumber =  [[${pageNumber}]];
    var urlPre =  "blog/";
    var GG = {
        "kk":function(mm){
```

```
                if(mm==1){
                    window.location.href = getRootPath_dc()+urlPre;
                }else{
                    window.location.href=  getRootPath_dc()+urlPre+mm;
                }
            }
        }
        $("#page").initPage(totalCount,pageNumber,GG.kk);

        function getRootPath_dc() {
            return window.location.protocol + '//' + window.location.host+"/";
        }

    </script>
    </html>
```

13.7.2　搜索页

搜索页大致内容与博客页相似，不过这里使用了复杂查询，所以需要特别提一下。首先需要在 Repository 层继承 JpaRepository 和 JpaSpecificationExecutor，如代码清单 13-19 所示。

代码清单 13-19　博客项目-搜索页数据操作层内容

```
    public interface ArticleRepository extends JpaRepository<Article, Long>,
JpaSpecificationExecutor<Article> {
    }
```

在 Service 层调用 JpaSpecificationExecutor 类中的 findAll 方法，这里需要传入 Specification 对象和 Pageable 对象。其中，Pageable 对象是分页对象，大家都很了解了；Specification 对象用于构建查询条件。

比如，文章要构建的语句如代码清单 13-20 所示。

代码清单 13-20　博客项目-搜索页查询 SQL

```
    SELECT
        article_id,
        article_authors,
        article_content,
        article_input_date,
        article_name,
        article_reading_time,
        is_enable,
        is_top
    FROM
        article
```

```
WHERE
    ( article_name LIKE '%第六%' OR article_content LIKE '%第六%' )
    AND is_enable = 1
ORDER BY
    article_id DESC
    LIMIT 10
```

这里创建一个 getWhereClause 方法用于构建查询条件，构建结果与上面 SQL 代码中 Where 后的语句一样，如代码清单 13-21 所示。

代码清单 13-21　博客项目-搜索页构建查询条件内容

```
private Specification<Article> getWhereClause(String keyword) {
    return new Specification<Article>() {
    @Override
    public Predicate toPredicate(Root<Article> root, CriteriaQuery<?> query,
CriteriaBuilder cb) {
        List<Predicate> predicate = new ArrayList<>();
        if (StringUtils.isNotBlank(keyword)) {
            predicate.add(
                cb.and(
                    cb.or(
                        cb.like(root.get("articleName"), "%" + keyword + "%"),
                        cb.like(root.get("articleContent"), "%" + keyword + "%")
                    )
                )
            );
        }
        predicate.add(cb.equal(root.get("isEnable"), Constants.YES));
        Predicate[] pre = new Predicate[predicate.size()];
        return query.where(predicate.toArray(pre)).getRestriction();
        }
    };
}
```

对应 Service 层调用如代码清单 13-22 所示。

代码清单 13-22　博客项目-搜索页 Service 层内容

```
@Override
public Page<Article> findSearchArticleList(int page, int size, String
keyword) {
    Pageable pageable = PageRequest.of(page, size, Sort.Direction.DESC,
"articleId");
    return articleRepository.findAll(this.getWhereClause(keyword),
pageable);
}
```

控制层及页面渲染数据与博客列表页类似，这里不再赘述。

13.7.3　文章详情页

文章详情页其实就是根据文章 ID 查询文章详情，在 Repository 层传入 articleId 和 isEnable，如代码清单 13-23 所示。

代码清单 13-23　博客项目-文章详情页数据操作层内容

```
Article findByIsEnableAndArticleId(Integer isEnable,Long articleId);
```

Service 层没有特别的东西，就是调用 Repository 层，如代码清单 13-24 所示。

代码清单 13-24　博客项目-文章详情页 Service 层内容

```
@Override
public Article findIsEnableArticleByArticleId(Long articleId) {
    return articleRepository.findByIsEnableAndArticleId(Constants.YES,
articleId);
}
```

Controller 层除了调用 Service 层查询数据之外，还需要更新文章阅读次数和使用 Markdown 转换工具将 articleContent 转换为可展示类型。完整 ArticleController 内容如代码清单 13-25 所示。

代码清单 13-25　博客项目-文章详情页 Controller 层内容

```
@Controller
public class ArticleController {

    @Autowired
    private ArticleService articleService;

    @GetMapping("/article/{id}")
    public String viewArticle(Model model, @PathVariable Long id) {
        Article article = articleService.findIsEnableArticleByArticleId(id);
        article.setArticleReadingTime(article.getArticleReadingTime() + 1);
        articleService.saveArticle(article);
        article.setArticleShowContent(MarkdownToHtml.markDownToHtml
(article.getArticleContent()));
        model.addAttribute("article", article);
        return "article";
    }
}
```

页面展示渲染数据都是使用 Thymeleaf 语言进行展示的，如代码清单 13-26 所示。

代码清单 13-26　博客项目-文章详情页前端渲染内容

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">
<html lang="en">

<head th:include="common/common_head::commonHeader"></head>

<body>

<div th:include="common/common_navigation :: commonNavigation"></div>
    <div id="white">
        <div class="container">
            <div class="row">
                <div class="col-lg-8 col-lg-offset-2">
                    <p><img th:src="@{/img/user.png}" width="50px"
height="50px"> <span th:text="${article.articleAuthors}"></span> </p>
                    <p >
                        <i class="fa fa-clock-o"></i><a th:text=
"${article.articleInputDate}"></a>
                        <i class="fa fa-eye"></i><a th:text=
"${article.articleReadingTime}"></a>
                    </p>
                    <h4 th:text="${article.articleName}"></h4>
                    <div th:utext="${article.articleShowContent}" style=
"overflow:hidden" > </div>
                    <p>标签:
                        <span th:each="al,iterStat : ${article.tagList}">
                         <i class="fa fa-tag"></i><a  target="_blank" th:href=
"${'/tag/'+al.tagName}" th:text="${al.tagName}"></a>
                        </span>
                    </p>
                    <hr>
                    <p><a th:href="@{/blog}"># 返回博客列表</a></p>
                </div>

            </div>
        </div>
    </div>
    <div th:include="common/common_footer :: commonFooter"></div>

  </body>
  <div th:include="common/common_onload_js :: onLoadJs"></div>

</html>
```

13.7.4　联系页

联系页的后台内容比较简单，后台使用 Message 对象接收前台传送的数据，并且在保存方法上加入了事务注解@Transactional。ContactController 类完整内容如代码清单 13-27 所示。

代码清单 13-27　博客项目-联系页 Controller 层内容

```
@Controller
public class ContactController {

    @GetMapping("/contact")
    public String contact() {
        return "contact";
    }

    @Autowired
    private MessageService messageService;

    @PostMapping("/contact/sendMail")
    @ResponseBody
    @Transactional(rollbackFor = Throwable.class)
    public String sendMail(@RequestBody Message message) {
        message.setMessageInputDate(new Date());
        messageService.saveMessage(message);
        return "success";
    }
}
```

Service 层和 Repository 层只是简单地调用和保存，这里不再赘述。接下来介绍前端发起 Ajax 请求，如代码清单 13-28 所示。

代码清单 13-28　博客项目-联系页前端 Ajax 方法内容

```
$("button").click(function(){
    if(!$('#contact_form').valid()){
        return false;
    }
    var url = getRootPath_dc() + "contact/sendMail";
    var name = $('#name').val();
    var email = $('#email').val();
    var subject = $('#subject').val();
    var messageContent = $('#messageContent').val();
    $.ajax({
        type : "POST",
        url : url,
```

```
        contentType: "application/json;charset=UTF-8",
        data : JSON.stringify({
            "name": name,
            "email": email,
            "subject": subject,
            "messageContent": messageContent
        }),
        success : function(result) {
            if(result == 'success'){
                alert("提交成功！");
            }else{
                alert("留言失败，请联系博客管理员。");
            }
        }
    });
});
```

主要功能大致介绍到这里。其他类似功能相信读者经过阅读源代码可以直接看懂，这里暂且略过。

13.8 辅助功能

本案例中的辅助功能其实并不是很多，有拦截器、定时器和初始化数据方法。接下来分别进行介绍。

13.8.1 拦截器

案例中拦截器的主要作用是加载底部数据和更新网站访问次数，完整内容如代码清单 13-29 所示。

代码清单 13-29　博客项目-拦截器内容

```
@Component
public class RequestInterceptor extends HandlerInterceptorAdapter {
    Logger logger = LoggerFactory.getLogger(RequestInterceptor.class);
    @Autowired
    private WebsiteAccessService websiteAccessService;
    @Autowired
    private TagService tagService;
    @Autowired
    private LinkService linkService;
```

```
    @Autowired
    private WebsiteConfigService websiteConfigService;
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        if(modelAndView != null){
            ModelMap modelMap = modelAndView.getModelMap();
            logger.info("正在更新网站访问次数。");
            WebsiteAccess websiteAccess = websiteAccessService.
getByAccessDateIs(new Date());
            websiteAccess.setAccessCount(websiteAccess.getAccessCount()+1);
            websiteAccessService.save(websiteAccess);
            logger.info("加入底部数据。");
            //标签列表
            modelMap.addAttribute("tagList", tagService.findAll());
            //友情链接列表
            modelMap.addAttribute("linkList",linkService.findAllByIsEnable());
            //友情链接列表
            modelMap.addAttribute("websiteConfig", websiteConfigService.
findWebsiteConfig());
        }
    }
}
```

13.8.2　定时器

案例中定时器使用 Spring 定时任务，主要用于在设定时间插入每日访问统计的记录，完整内容如代码清单 13-30 所示。

代码清单 13-30　博客项目-定时器内容

```
@Component
public class WebSiteTimer {
    @Autowired
    private WebsiteAccessService websiteAccessService;
    @Scheduled(cron = "0 0 0 1/1 * ?")
    private void updateTodayWebsiteVisits() {
        WebsiteAccess websiteAccess = new WebsiteAccess();
        websiteAccess.setAccessCount(1);
        websiteAccess.setAccessDate(new Date());
        websiteAccessService.save(websiteAccess);
    }
}
```

13.8.3 初始化

这个方法只是在没有数据测试的时候报错才添加的，是在拦截器更新网站访问次数的时候，website_access 表内没有数据引起的。所以这里设置了一个方法，查询当天是否存在 website_access 表的数据，如果不存在，就插入一条。博客配置表也存在同样的问题，不存在数据的话，需要插入一条默认数据来避免这个问题。当然，读者也可以选择别的方式处理这个问题。完整内容如代码清单 13-31 所示。

代码清单 13-31　博客项目-初始化数据方法内容

```
@Component
public class InitWebsiteData {
    @Autowired
    private WebsiteAccessService websiteAccessService;

    @Autowired
    private WebsiteConfigService websiteConfigService;

    @PostConstruct
    public void initWebsiteVisits(){
        //查询当日是否存在博客访问表记录，若不存在，则插入
        if(websiteAccessService.getByAccessDateIs(new Date()) == null){
            WebsiteAccess websiteAccess = new WebsiteAccess();
            websiteAccess.setAccessCount(1);
            websiteAccess.setAccessDate(new Date());
            websiteAccessService.save(websiteAccess);
        }
        //查询当日是否存在博客配置表记录，若不存在，则插入
        if(websiteConfigService.findWebsiteConfig()==null){
            WebsiteConfig websiteConfig = new WebsiteConfig();
            websiteConfig.setId(1L);
            websiteConfig.setAboutPageArticleId(7L);
            websiteConfig.setBlogName("SpringBoot 博客");
            websiteConfig.setAuthorName("dalaoyang");
            websiteConfig.setDomainName("Dalaoyang.cn");
            websiteConfig.setRecordNumber("辽 ICP 备 17014944 号-1");
            websiteConfig.setEmailUsername("dalaoyang@aliyun.com");
            websiteConfigService.saveWebsiteConfig(websiteConfig);
        }
    }
}
```

辅助功能到这里就大致结束了。其实博客还有很多地方可以扩展，读者可以自由发挥。

13.9 小 结

本章介绍了如何使用之前学习的内容构建一个博客系统，建议读者在笔者制作的 Thymeleaf 模板的基础上进行练习。

另外，笔者写了一个 SQL 脚本供大家预设数据进行测试，如代码清单 13-32 所示。

代码清单 13-32 博客项目-初始化内容 SQL

```
## init article
INSERT INTO `springbootBlog`.`article`(`article_id`, `article_authors`, `article_content`, `article_input_date`, `article_name`, `article_reading_time`, `is_enable`, `is_top`) VALUES (1, 'dalaoyang', '这是第一篇博客的摘要。这是第一篇博客的摘要。这是第一篇博客的摘要。这是第一篇博客的摘要。这是第一篇博客的摘要。这是第一篇博客的摘要。这是第一篇博客的摘要。', '2019-01-01 00:00:00', '第一篇博客', 1, 1, 1);
INSERT INTO `springbootBlog`.`article`(`article_id`, `article_authors`, `article_content`, `article_input_date`, `article_name`, `article_reading_time`, `is_enable`, `is_top`) VALUES (2, 'dalaoyang', '这是第二篇博客的摘要。这是第二篇博客的摘要。这是第二篇博客的摘要。这是第二篇博客的摘要。这是第二篇博客的摘要。这是第二篇博客的摘要。这是第二篇博客的摘要。', '2019-01-01 00:00:00', '第二篇博客', 1, 1, 1);
INSERT INTO `springbootBlog`.`article`(`article_id`, `article_authors`, `article_content`, `article_input_date`, `article_name`, `article_reading_time`, `is_enable`, `is_top`) VALUES (3, 'dalaoyang', '这是第三篇博客的摘要。这是第三篇博客的摘要。这是第三篇博客的摘要。这是第三篇博客的摘要。这是第三篇博客的摘要。这是第三篇博客的摘要。这是第三篇博客的摘要。', '2019-01-01 00:00:00', '第三篇博客', 1, 1, 1);
INSERT INTO `springbootBlog`.`article`(`article_id`, `article_authors`, `article_content`, `article_input_date`, `article_name`, `article_reading_time`, `is_enable`, `is_top`) VALUES (4, 'dalaoyang', '这是第四篇博客的摘要。这是第四篇博客的摘要。这是第四篇博客的摘要。这是第四篇博客的摘要。这是第四篇博客的摘要。这是第四篇博客的摘要。这是第四篇博客的摘要。', '2019-01-01 00:00:00', '第四篇博客', 1, 1, 1);
INSERT INTO `springbootBlog`.`article`(`article_id`, `article_authors`, `article_content`, `article_input_date`, `article_name`, `article_reading_time`, `is_enable`, `is_top`) VALUES (5, 'dalaoyang', '这是第五篇博客的摘要。这是第五篇博客的摘要。这是第五篇博客的摘要。这是第五篇博客的摘要。这是第五篇博客的摘要。这是第五篇博客的摘要。这是第五篇博客的摘要。', '2019-01-01 00:00:00', '第五篇博客', 1, 1, 1);
```

```
INSERT INTO `springbootBlog`.`article`(`article_id`, `article_authors`, `article_content`, `article_input_date`, `article_name`, `article_reading_time`, `is_enable`, `is_top`) VALUES (6, 'dalaoyang', '这是第六篇博客的摘要。这是第六篇博客的摘要。这是第六篇博客的摘要。这是第六篇博客的摘要。这是第六篇博客的摘要。这是第六篇博客的摘要。这是第六篇博客的摘要。', '2019-01-01 00:00:00', '第六篇博客', 1, 1, 1);
INSERT INTO `springbootBlog`.`article`(`article_id`, `article_authors`, `article_content`, `article_input_date`, `article_name`, `article_reading_time`, `is_enable`, `is_top`) VALUES (7, 'dalaoyang', '大家好，我是 Spring Boot2 实战之旅的作者杨洋，感谢大家对我的支持，谢谢。\n\n', '2019-02-21', 'About DALAOYANG!\n\n', 0, 0, 0);
## init link
INSERT INTO `springbootBlog`.`link`(`link_id`, `link_name`, `link_url`, `remark`) VALUES (1, '简书', 'https://www.jianshu.com/u/128b6effde53', '简书地址');
INSERT INTO `springbootBlog`.`link`(`link_id`, `link_name`, `link_url`, `remark`) VALUES (2, 'DALAOYANG', 'https://www.dalaoyang.cn', 'dalaoyang 的博客');
## init tag
INSERT INTO `springbootBlog`.`tag`(`tag_id`, `tag_name`) VALUES (1, 'SpringBoot');
INSERT INTO `springbootBlog`.`tag`(`tag_id`, `tag_name`) VALUES (2, 'SpringCloud');
INSERT INTO `springbootBlog`.`tag`(`tag_id`, `tag_name`) VALUES (3, 'Nginx');
INSERT INTO `springbootBlog`.`tag`(`tag_id`, `tag_name`) VALUES (4, 'Linux');
INSERT INTO `springbootBlog`.`tag`(`tag_id`, `tag_name`) VALUES (5, 'Tomcat');
INSERT INTO `springbootBlog`.`tag`(`tag_id`, `tag_name`) VALUES (6, 'Java');
## init article_tag
INSERT INTO `springbootBlog`.`article_tag`(`article_id`, `tag_id`) VALUES (1, 1);
INSERT INTO `springbootBlog`.`article_tag`(`article_id`, `tag_id`) VALUES (1, 3);
INSERT INTO `springbootBlog`.`article_tag`(`article_id`, `tag_id`) VALUES (2, 3);
INSERT INTO `springbootBlog`.`article_tag`(`article_id`, `tag_id`) VALUES (2, 6);
INSERT INTO `springbootBlog`.`article_tag`(`article_id`, `tag_id`) VALUES (3, 1);
INSERT INTO `springbootBlog`.`article_tag`(`article_id`, `tag_id`) VALUES (4, 2);
```

```
INSERT INTO `springbootBlog`.`article_tag`(`article_id`, `tag_id`) VALUES
(5, 1);
INSERT INTO `springbootBlog`.`article_tag`(`article_id`, `tag_id`) VALUES
(6, 2);

## maybe error----------
##init website_config
INSERT INTO `springbootBlog`.`website_config`(`id`,
`about_page_article_id`, `author_name`, `blog_name`, `email_username`,
`domain_name`, `record_number`) VALUES (1, 7, 'dalaoyang', 'SpringBoot 博客',
'smtp.aliyun.com', 'Dalaoyang.cn', '辽 ICP 备 17014944 号-1');
##init website_access
INSERT INTO `springbootBlog`.`website_access`(`id`, `access_count`,
`access_date`) VALUES (1, 0, now());
```

第 14 章

Spring Boot 实战之博客后台系统

第 13 章介绍了如何利用 Spring Boot 制作博客，但是只有一个博客系统，每次发布文章时都需要手动向数据库插入数据，这样显然有些麻烦。本章将带领读者结合第 13 章的博客创建一个博客后台系统。

14.1 博客后台的制作思路

博客后台系统用于维护博客的一些相关信息，如文章的管理、标签的管理、友情链接的管理及博客数据的统计等，制作思路与制作博客系统一致。我们回顾一下制作思路：

（1）静态模板项目制作，将 HTML 静态项目改为 Thymeleaf 项目，使用 Controller 进行跳转。

（2）实体设计，因为使用的是 Spring Data JPA，所以实体设计决定着数据库表的结构。

（3）后台方法代码编写，包含查询数据库、封装数据等。

（4）渲染数据，将后台查询出来的数据动态渲染到 Thymeleaf。

14.1.1 博客后台布局介绍

案例博客布局分为 4 部分，其中头部、底部和左侧导航部分是公用部分，右侧为根据左侧导航动态显示的内容。4 部分分别说明如下。

- 头部：头部左侧为博客后台系统名称，右侧有 3 个功能，铃铛图标显示当日博客的内容信息，包括有几条未读消息、几篇新增文章、几个新增标签；邮件图标显示最近有几条未读消息；最右侧为退出登录按钮。

- 左侧导航：左侧导航分为 7 个模块，分别是首页、文章管理、标签管理、友情链接管理、用户管理、消息管理、系统管理。其中，文章管理、标签管理、友情链接管理和用户管理包含二级菜单，如图 14-1 所示。

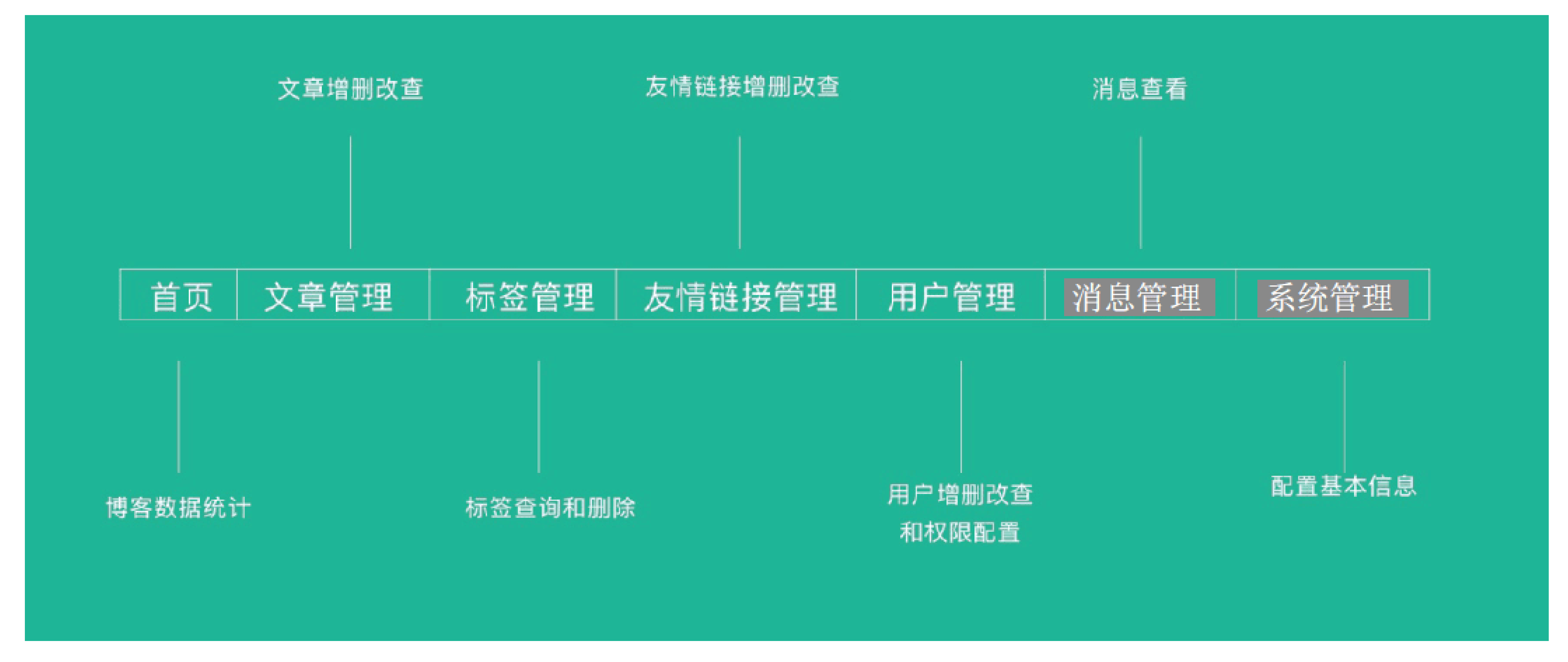


图 14-1 左侧导航结构图

- 右侧内容：右侧显示模块对应的内容，为各个模块或功能显示的内容，稍后会详细介绍。
- 底部：底部只显示一些博客配置信息，如域名和备案号，比较简单。

14.1.2 博客功能介绍

博客功能部分就是图 14-1 中单击导航显示的对应内容，分为以下几个功能。

（1）首页：首页主要是统计一些文章的信息，如文章数量统计、标签数量统计、友情链接数量统计、消息数量统计、当日访问量、本周访问量、当月访问量、总访问量和近 10 日访问统计图表。

（2）文章管理：文章管理分为两个子菜单：文章列表和新增文章。文章列表页可以查询文章、预览文章（需要跳转到博客系统），单击“修改文章”跳转到修改文章页面，可以启动和禁用文章、删除文章。新增文章页用于增添文章和修改文章。

（3）标签管理：标签管理中含有标签列表页子菜单。标签列表页可以查询标签和删除标签。

（4）友情链接管理：友情链接管理包含友情链接列表和新增友情链接，友情链接列表页包含查询友情链接、删除友情链接和修改友情链接。新增友情链接页用于新增和修改友情链接。

（5）用户管理：用户管理中含有用户列表和新增用户。用户列表页可以查询用户、修改用户和禁用用户。新增用户页用于新增和修改用户。

（6）消息管理：消息管理用于查看消息列表和查询某条消息详情。

（7）系统管理：系统管理用于修改博客系统配置信息。

14.2 博客后台模板制作

博客后台模板制作与第 13 章一致，大体思路就是将一些通用的配置和模块提取出来，这里不再赘述，读者可以根据第 13 章的提取方法进行博客后台模板的制作。

14.3 效 果 展 示

本节来看一下博客后台的效果图。

登录页如图 14-2 所示。



图 14-2 登录页效果图

首页如图 14-3 所示。

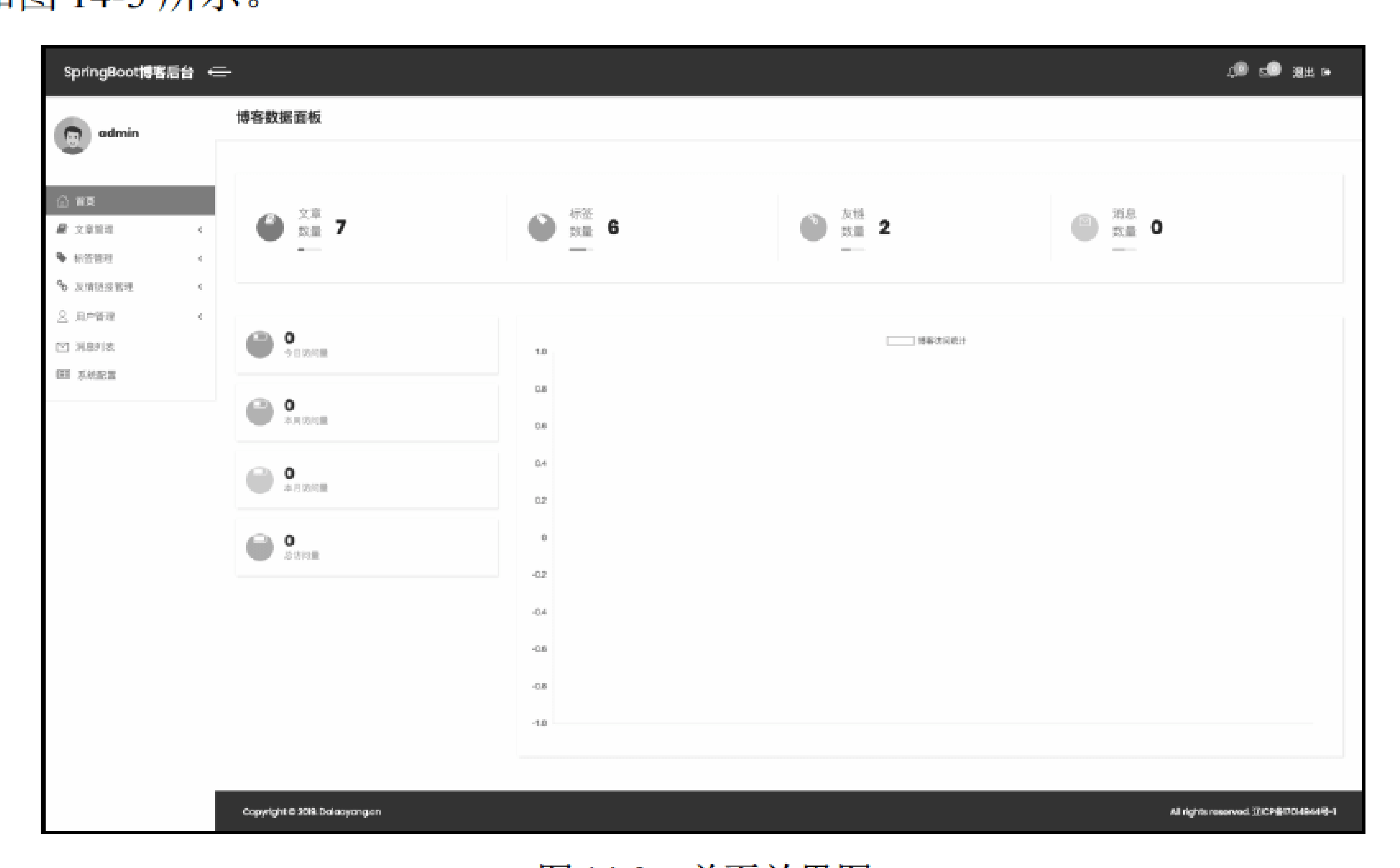


图 14-3 首页效果图

文章列表页如图 14-4 所示。

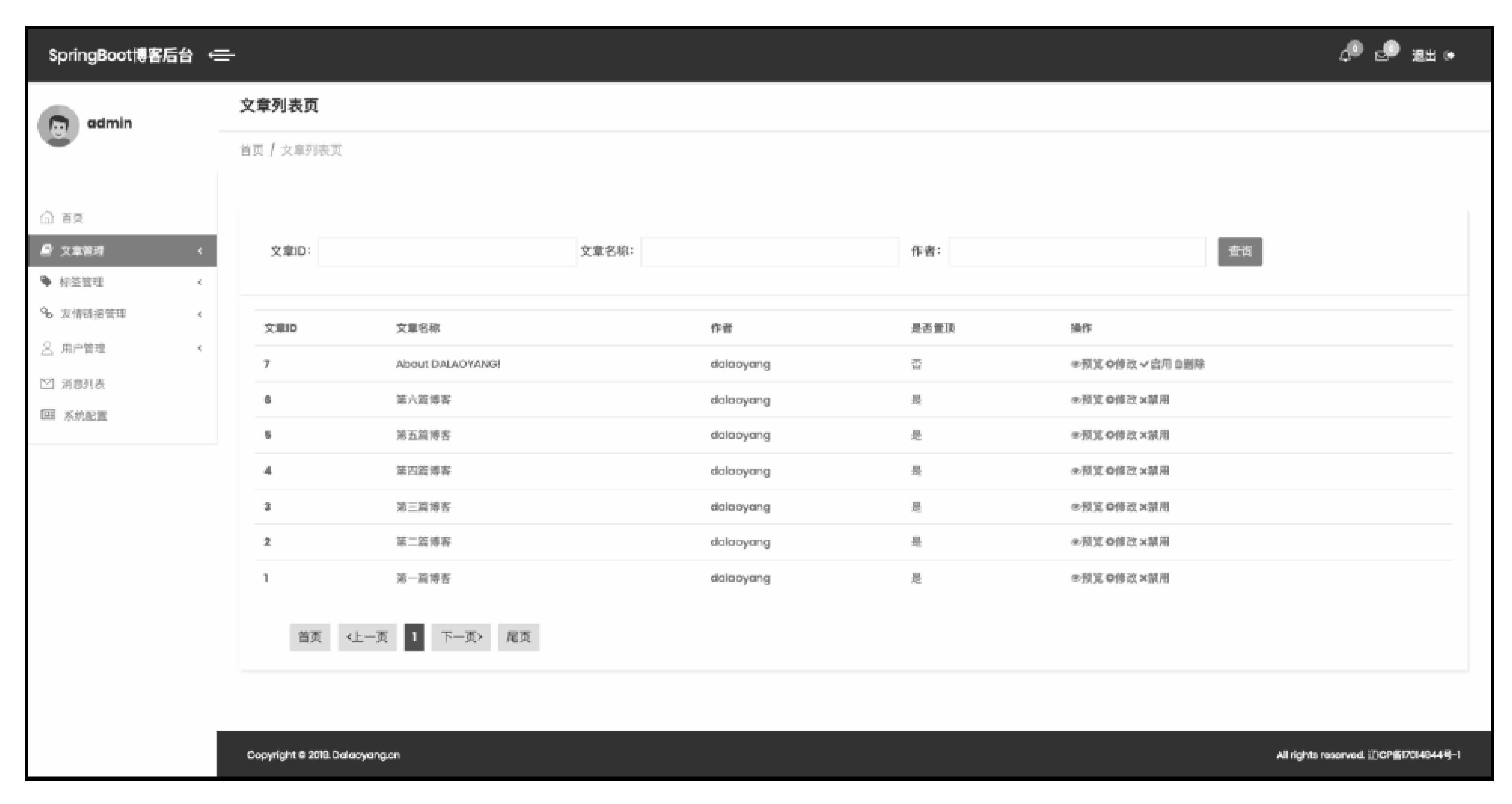


图 14-4　文章列表页效果图

文章编辑页如图 14-5 所示。

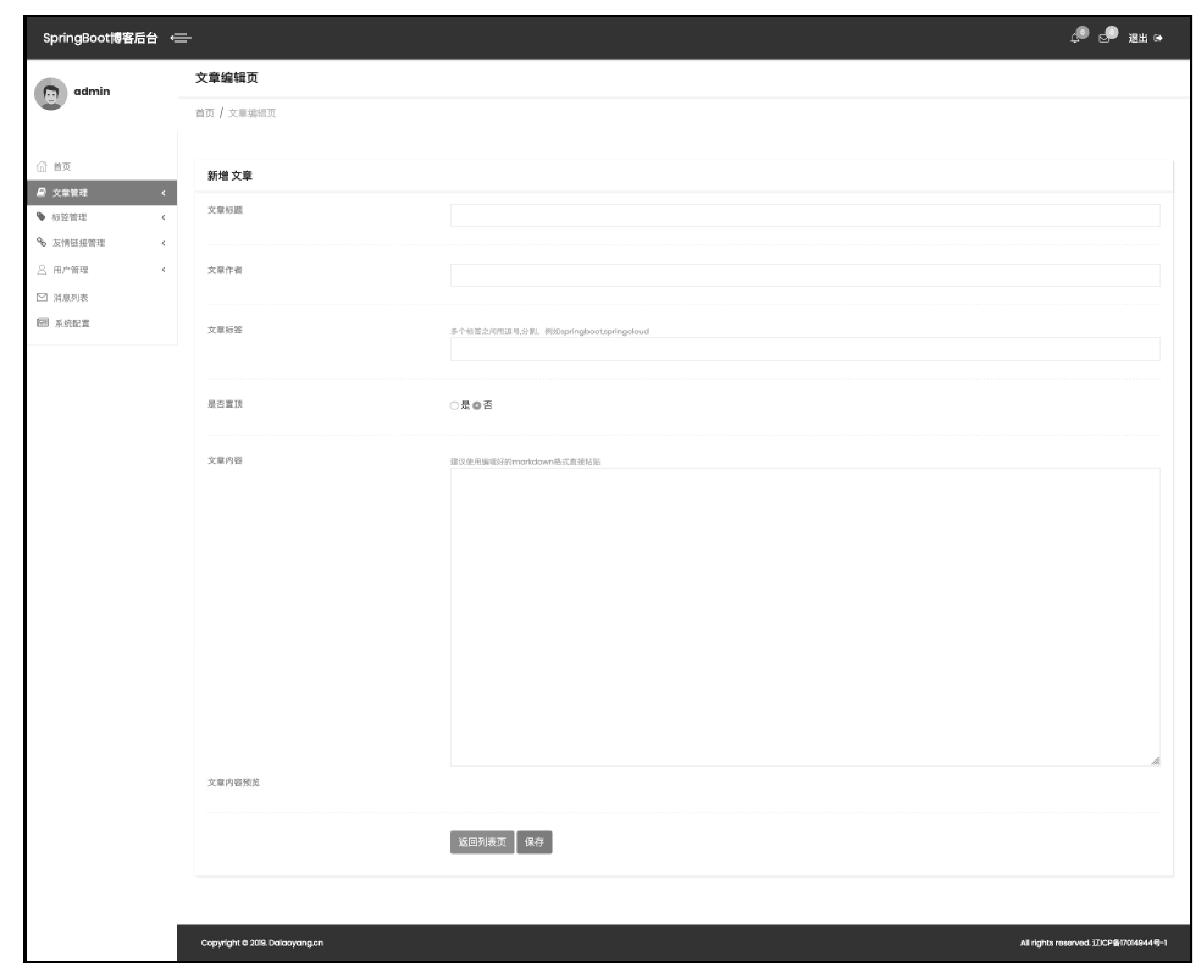


图 14-5　文章编辑页效果图

标签列表页如图 14-6 所示。

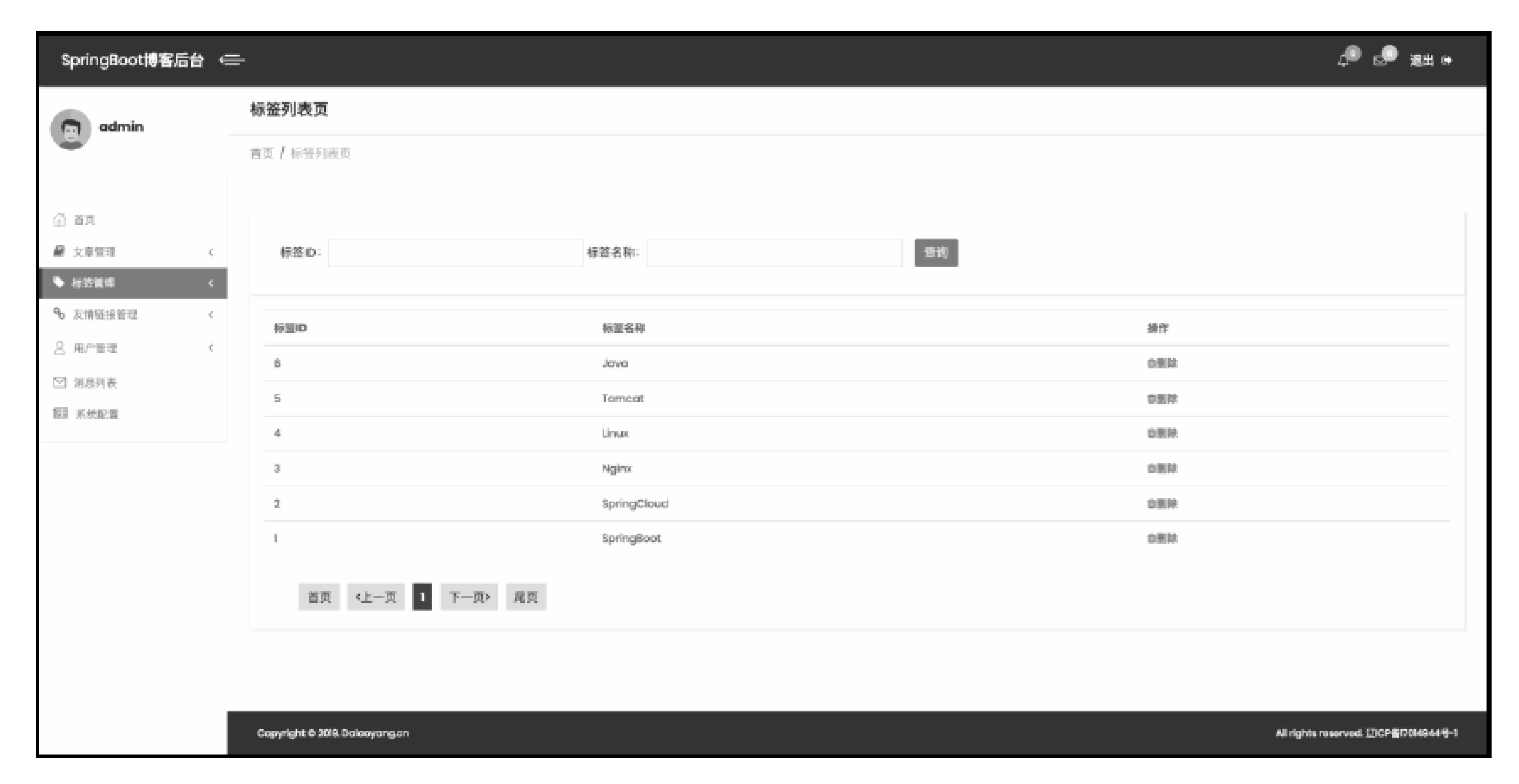


图 14-6　标签列表页效果图

友情链接列表页如图 14-7 所示。

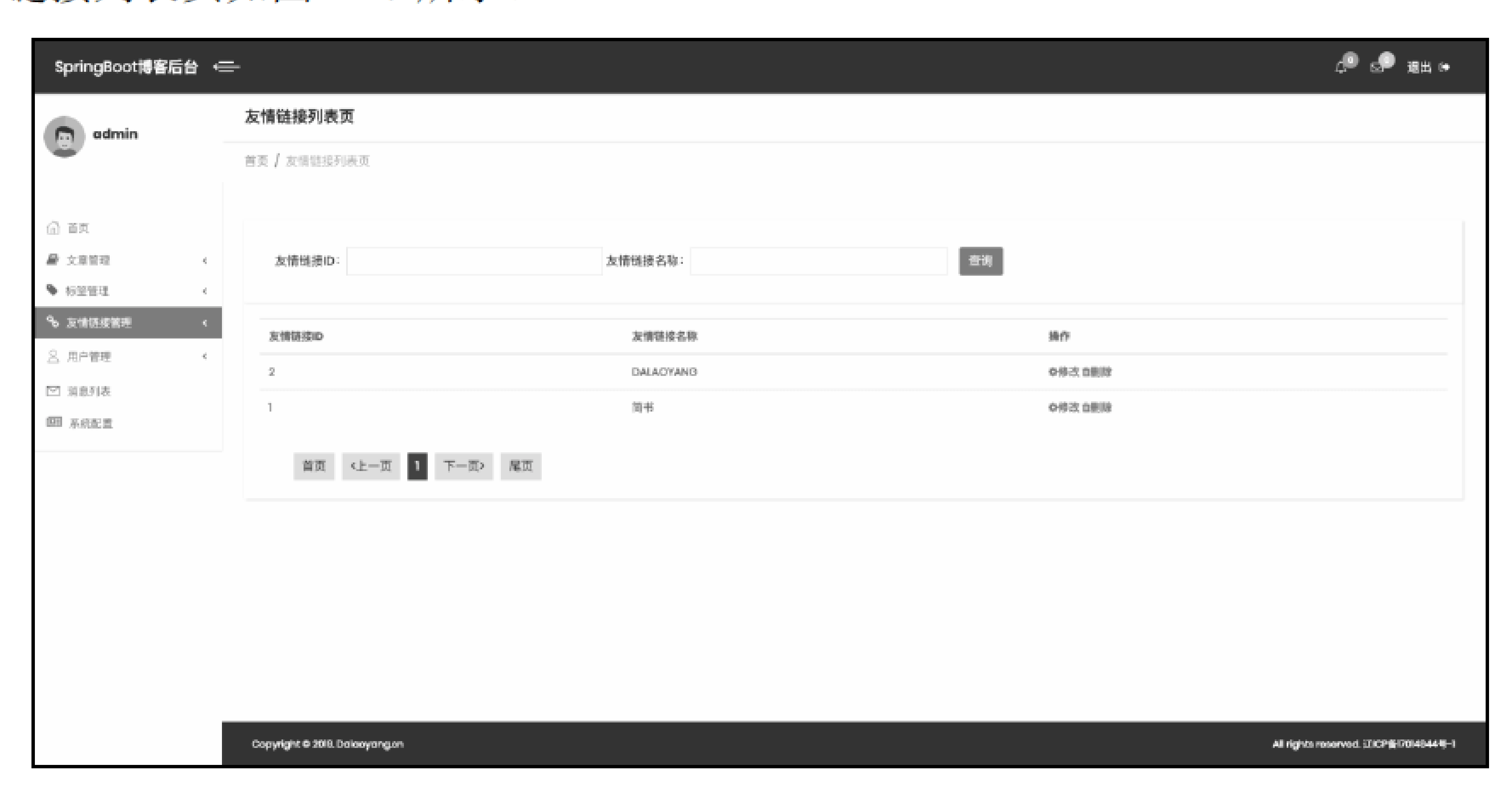


图 14-7　友情链接列表页效果图

友情链接编辑页如图 14-8 所示。

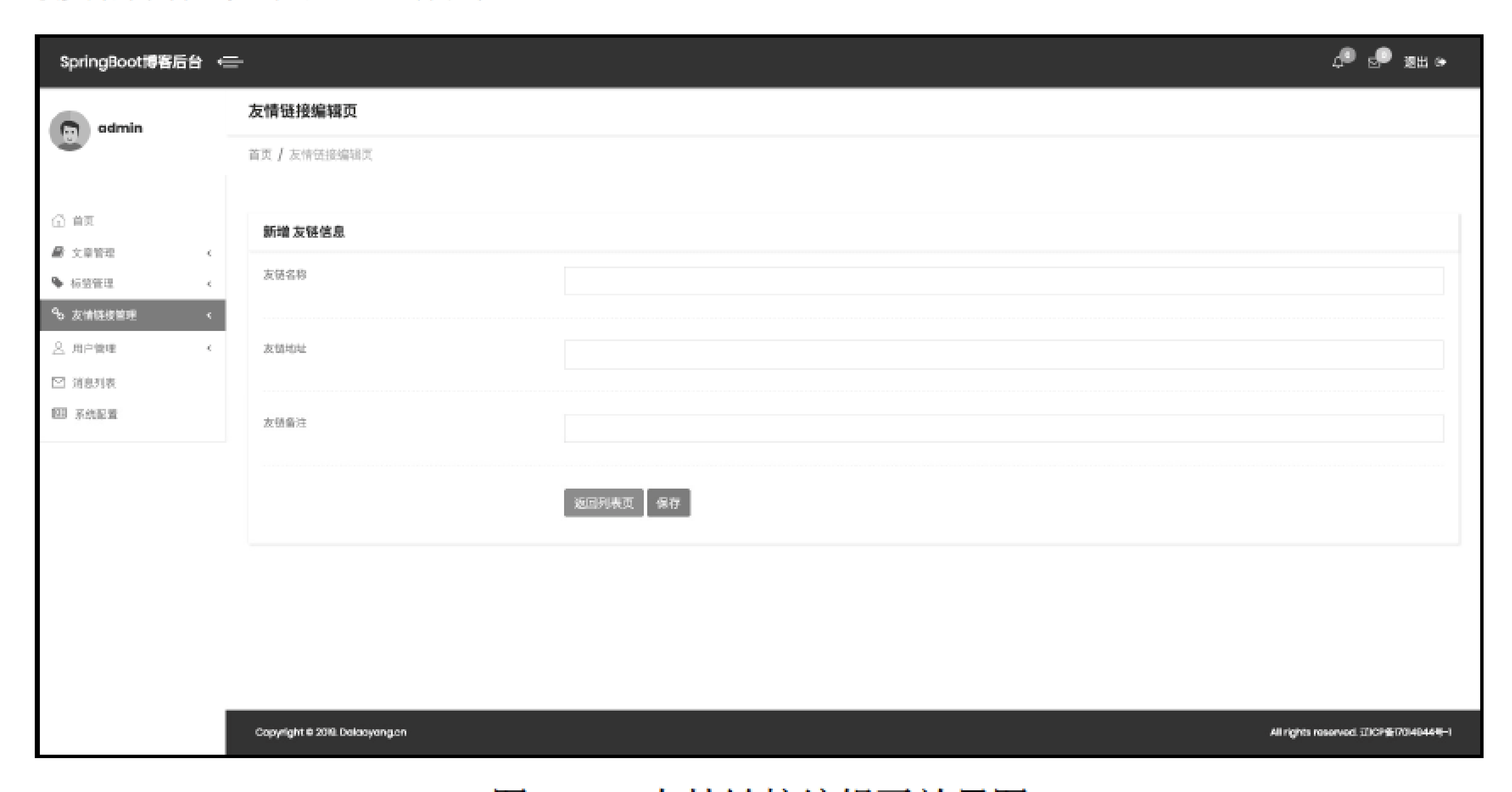


图 14-8　友情链接编辑页效果图

用户列表页如图 14-9 所示。

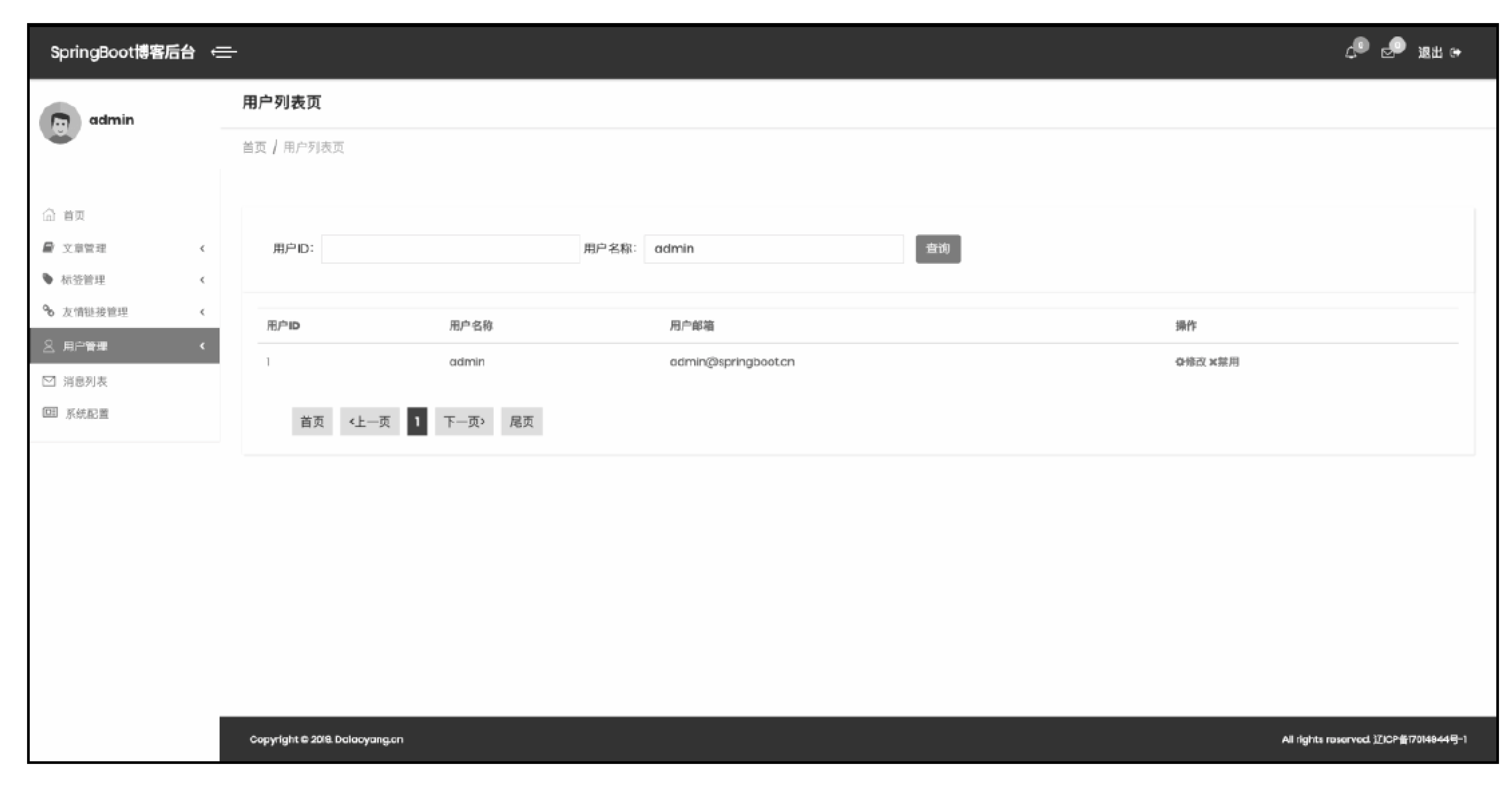


图 14-9　用户列表页效果图

用户编辑页如图 14-10 所示。

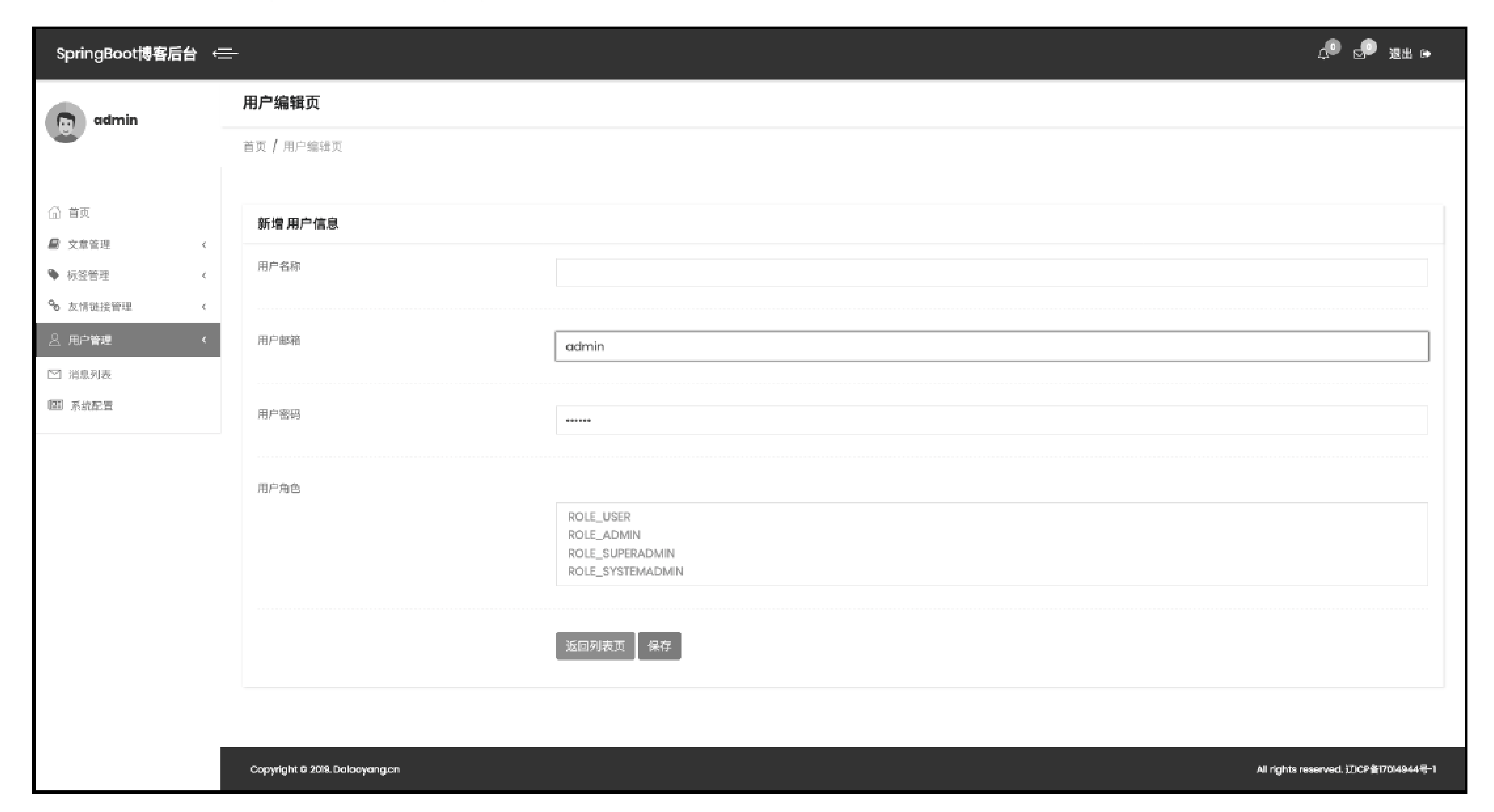


图 14-10　用户编辑页效果图

消息列表页如图 14-11 所示。



图 14-11　消息列表页效果图

系统配置页如图 14-12 所示。

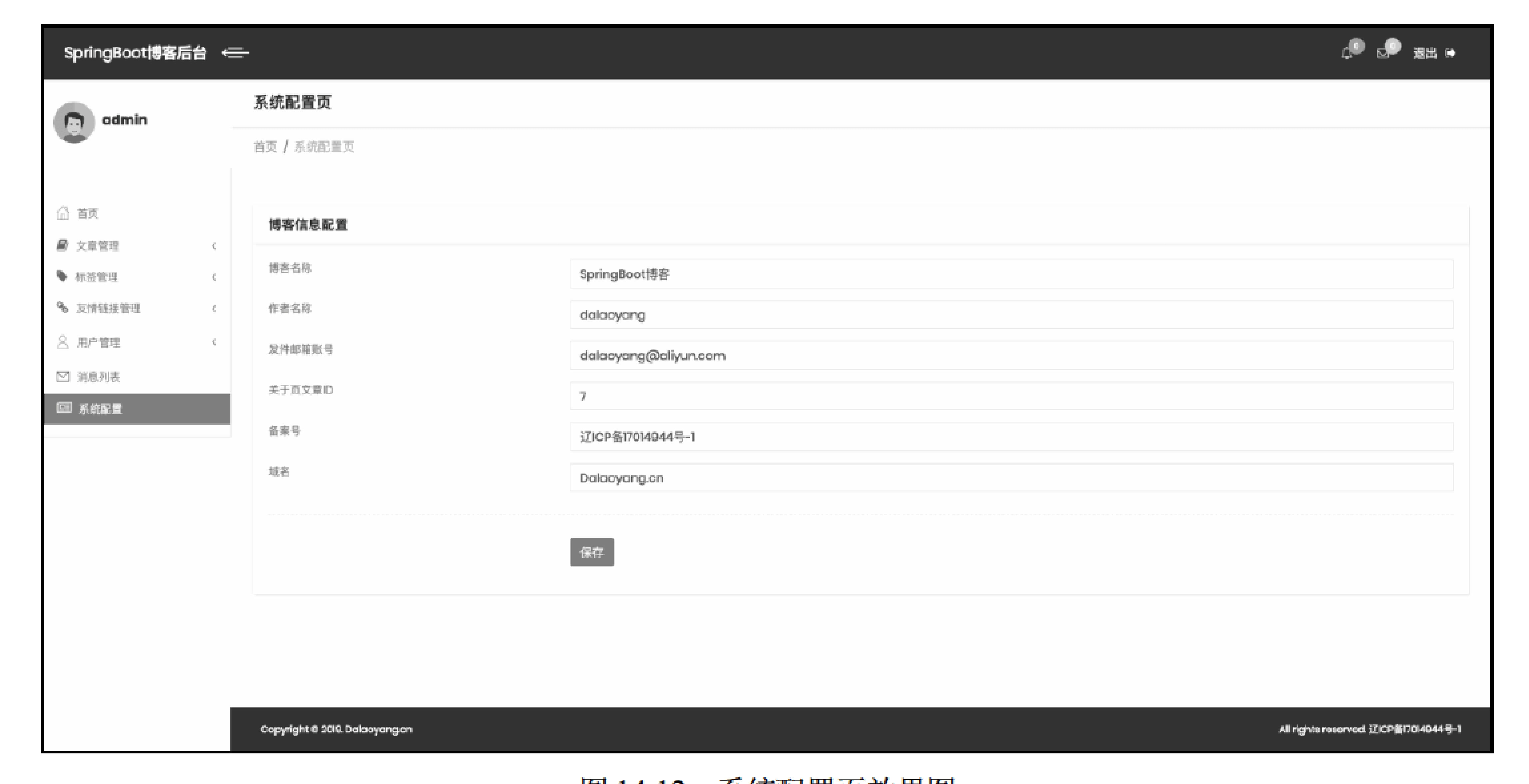


图 14-12　系统配置页效果图

14.4　依赖配置

依赖配置与博客系统大致相同，不过这里加入了 spring-boot-starter-mail 依赖进行邮件发送、spring-boot-starter-security 依赖进行登录和授权。完整依赖内容如代码清单 14-1 所示。

代码清单 14-1　博客后台项目-依赖文件内容

```
<dependencies>
    <!-- jpa-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <!-- thymeleaf-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <!-- web-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- mysql-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <!-- 去除 thymeleaf 严格校验-->
    <dependency>
        <groupId>net.sourceforge.nekohtml</groupId>
        <artifactId>nekohtml</artifactId>
        <version>1.9.22</version>
    </dependency>
    <!-- lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.20</version>
    </dependency>
    <!-- mail-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>
    <!-- security-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <!-- commons-lang-->
        <dependency>
            <groupId>commons-lang</groupId>
            <artifactId>commons-lang</artifactId>
            <version>2.6</version>
        </dependency>
        <!-- commons-collections-->
        <dependency>
            <groupId>commons-collections</groupId>
            <artifactId>commons-collections</artifactId>
            <version>3.2.2</version>
        </dependency>
    </dependencies>
```

14.5 配 置 文 件

配置文件与博客系统的配置大致相同，只不过端口号使用的是 10001。这里需要提醒一下，spring.jpa.hibernate.ddl-auto 属性没有特殊需求的话，尽量设置为 update 或者 none，如果设置成 create，就会造成数据丢失，并且这里加入了邮箱相关配置，在后台系统中设置了定时器，将每日的数据发送至邮箱内（邮箱配置需要修改成自己的邮箱配置）。完整配置如代码清单 14-2 所示。

代码清单 14-2　博客后台项目-依赖文件内容

```
    ##端口号
    server.port=10001
    ##禁用 thymeleaf 缓存
    spring.thymeleaf.cache=false

    ##数据库配置
    ##数据库地址
    spring.datasource.url=jdbc:mysql://localhost:3306/springbootBlog?characte
rEncoding=utf8&useSSL=false
    ##数据库用户名
    spring.datasource.username=root
    ##数据库密码
    spring.datasource.password=root
    ##数据库驱动
    spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

```
##none  启动时不做任何操作
spring.jpa.hibernate.ddl-auto=update
##控制台打印 sql
spring.jpa.show-sql=true

##邮箱服务器 smtp 地址
spring.mail.host=邮箱服务器 smtp 地址
##邮箱用户名
spring.mail.username=邮箱名称
##邮箱密码（注意：qq 邮箱应该使用独立密码，在 qq 邮箱设置中获取）
spring.mail.password=邮箱密码
##编码格式
spring.mail.default-encoding=UTF-8
```

14.6 后 台 实 体

由于后台系统与第 13 章的博客系统共用一套实体，因此这里不再介绍实体。本节介绍博客后台系统独立的表。

14.6.1 用户表

用户表用于存储博客后台系统的用户信息，表名为 user。针对当前案例，设置了如下几个字段属性。

- userId：用户表主键 ID，设置了主键自增列。
- username：用户名称、登录名。
- password：用户密码。
- email：用户邮箱。
- isEnable：是否启用，1 为启用，0 为禁用。
- roleList：设置与角色表的多对多关系。

roleIdList 属性是项目中使用的，但是并非数据库字段。

- roleIdList：角色 ID 集合。

完整 user 实体类内容如代码清单 14-3 所示。

代码清单 14-3 博客后台项目-user 实体内容

```
@Entity
@Table(name = "user")
@Data
```

```
@AllArgsConstructor
@NoArgsConstructor
public class User implements Serializable {
    private static final long serialVersionUID = 3033545151355633270L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String username;
    private String password;
    private String email;
    private Integer isEnable;

    @ManyToMany(fetch=FetchType.EAGER)
    @JoinTable(name = "userRole", joinColumns = {@JoinColumn(name =
"userId")}, inverseJoinColumns = {@JoinColumn(name = "roleId")})
    private List<Role> roleList;

    @Transient
    private List<Long> roleIdList;

    public User(Long userId,String username, String password, String email,
Integer isEnable,List<Role> roleList) {
        this.userId = userId;
        this.username = username;
        this.password = password;
        this.email = email;
        this.isEnable = isEnable;
        this.roleList = roleList;
    }
}
```

14.6.2 角色表

角色表用于存储角色信息，表名为 role。针对当前案例，设置了如下几个字段属性。

- roleId：角色表主键 ID，设置了主键自增列。
- roleName：角色名称。
- isEnable：是否启用，1 为启用，0 为禁用。

完整 role 实体类内容如代码清单 14-4 所示。

代码清单 14-4 博客后台项目-role 实体内容

```
@Entity
```

```
@Table(name = "role")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Role implements Serializable {
    private static final long serialVersionUID = 3392729947020278189L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long roleId;
    private String roleName;
    private Integer isEnable;

    @Transient
    private Integer isHave;

    public Role(Long roleId,String roleName, Integer isEnable) {
        this.roleId = roleId;
        this.roleName = roleName;
        this.isEnable = isEnable;
    }
}
```

14.7 主　功　能

与第 13 章一样，首先来看应用程序的目录结构，如图 14-13 所示。

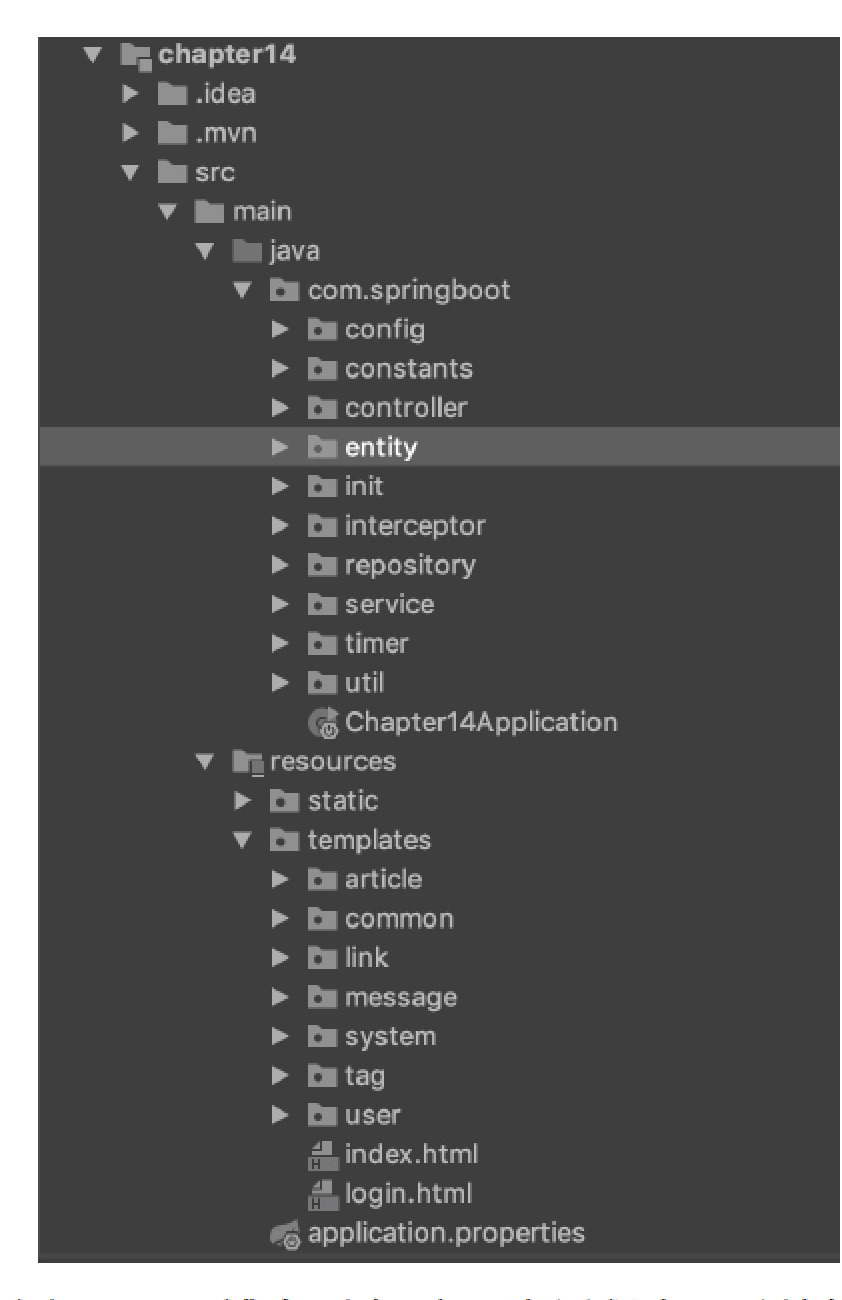


图 14-13　博客后台项目-应用程序目录结构

其中，每个包对应的功能如下。

- config：配置，案例中用于配置拦截器和 Spring Security 授权及认证配置。
- constants：常量，案例中常量类所在的包。
- controller：控制层，案例中控制层大多只用于封装数据和页面跳转。
- entity：实体类。
- init：初始化，用于初始化一些应用中的数据。
- interceptors：拦截器，案例中加载登录用户信息、头部信息等。
- repository：数据操作层，用于 JPA 操作数据库。
- service：业务层，案例中用于调用数据操作层和一些业务逻辑处理。

- timer：定时器，案例中用于通过邮件发送每日博客数据统计。
- util：工具，案例中只有一个计算日期的工具类。

接下来介绍两个典型的功能：首页和文章管理。

14.7.1 首页

在首页主要进行一些数据统计，比如简单汇总统计、文章统计、标签统计、友情链接统计和消息统计，这些都是利用 Repository 中默认的 count()方法实现的。另外，还有一些访问量统计，比如今日访问量、本周访问量、本月访问量和总访问量。在 JPA 中，如果需要使用聚合查询，就需要使用 CriteriaBuilder 来构建聚合查询，如代码清单 14-5 所示。

代码清单 14-5 博客后台项目-聚合函数数据统计

```
   @Override
   public Integer sumWebsiteAccess(Date date, Integer days) {
      CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
      CriteriaQuery<Integer> query = criteriaBuilder.createQuery
(Integer.class);
      Root<WebsiteAccess> root = query.from(WebsiteAccess.class);
      query.select(criteriaBuilder.sum(root.get("accessCount")));
      if (days == null && date != null) {
         Predicate predicate = criteriaBuilder.equal(root.get("accessDate"),
date);
         query.where(predicate);
      } else if (date != null) {
         Predicate predicate = criteriaBuilder.between(root.get
("accessDate"), DateUtils.getDateBefore(date, days), date);
            query.where(predicate);
      }
      Integer result = entityManager.createQuery(query).getSingleResult();
      if (result == null) {
         result = 0;
      }
      return result;
   }
```

方法查询的 SQL 如代码清单 14-6 所示。

代码清单 14-6 博客后台项目-聚合函数 SQL

```
   select sum(websiteacc0_.access_count) as col_0_0_ from website_access
websiteacc0_ where websiteacc0_.access_date=?
   select sum(websiteacc0_.access_count) as col_0_0_ from website_access
websiteacc0_ where websiteacc0_.access_date between ? and ?
```

```
    select sum(websiteacc0_.access_count) as col_0_0_ from website_access
websiteacc0_ where websiteacc0_.access_date between ? and ?
    select sum(websiteacc0_.access_count) as col_0_0_ from website_access
websiteacc0_
```

另外，首页还包含一个访问图标统计图，就是查询最后 10 天的访问记录表，不过在使用 charts 展示时需要将数量和日期分别从集合中提取出来。这里使用 Lambda 表达式提取对应属性，如代码清单 14-7 所示。

代码清单 14-7　博客后台项目-数据图表数据查询

```
    List<WebsiteAccess> websiteAccessList =
websiteAccessService.findChartsWebsiteAccess();
    List<Integer> websiteAccessCountList =
websiteAccessList.stream().map(WebsiteAccess::getAccessCount).
    collect(Collectors.toList());
    List<Date> websiteAccessDateList =
websiteAccessList.stream().map(WebsiteAccess::getAccessDate).
    collect(Collectors.toList());
```

完整首页加载数据查看源代码即可，很多内容之前已经讲过了。

14.7.2　文章管理

在博客后台系统中，很多列表的实现都很相近，这里以文章列表为例进行介绍。

1. 列表查询

文章列表查询其实就是一个带有条件和分页的复杂查询，这里再对复杂查询介绍一次。使用 Specification 对象构建查询条件，这里设置文章 ID 字段为精确查询，文章名称和作者名称都是使用右模糊查询。构建查询条件的方法如代码清单 14-8 所示。

代码清单 14-8　博客后台项目-文章管理构建查询条件内容

```
    private Specification<Article> getWhereClause(Long articleId, String
articleName, String articleAuthors) {
            return new Specification<Article>() {
                @Override
                public Predicate toPredicate(Root<Article> root, CriteriaQuery<?>
query, CriteriaBuilder cb) {
                    List<Predicate> predicate = new ArrayList<>();
                    if (articleId != null) {
                        predicate.add(
                                cb.or(cb.equal(root.get("articleId"), articleId))
                        );
```

```
                    }
                    if (!StringUtils.isEmpty(articleName)) {
                        predicate.add(
                                cb.or(cb.like(root.get("articleName"), articleName
+ "%"))
                        );
                    }
                    if (!StringUtils.isEmpty(articleAuthors)) {
                        predicate.add(
                                cb.or(cb.like(root.get("articleAuthors"),
articleAuthors + "%"))
                        );
                    }
                    Predicate[] pre = new Predicate[predicate.size()];
                    return query.where(predicate.toArray(pre)).getRestriction();
                }
            };
        }
```

Service 层调用如代码清单 14-9 所示。

代码清单 14-9　博客后台项目-文章管理 Service 层内容

```
    @Override
    public Page<Article> findAllBySearch(Pages pages, Long articleId, String
articleName, String articleAuthors) {
        Pageable pageable = PageRequest.of(pages.getPage(), pages.getPageSize(),
Sort.Direction.DESC, "articleId");
        return articleRepository.findAll(this.getWhereClause(articleId,
articleName, articleAuthors), pageable);
    }
```

Controller 层进行一些数据的封装，比如查询数据的返回、分页参数的返回、列表返回等。完整内容如代码清单 14-10 所示。

代码清单 14-10　博客后台项目-文章管理 Controller 内容

```
    @GetMapping("/list")
    public String article(Integer pageNumber, Long articleId, String articleName,
                          String articleAuthors, Model model) {
        Pages pages = Pages.defaultPages(pageNumber);
        Page<Article> articlePage = articleService.findAllBySearch(pages,
articleId, articleName, articleAuthors);
        model.addAttribute("articleList", articlePage.getContent());
        model.addAttribute("totalCount", articlePage.getTotalElements());
```

```
    model.addAttribute("pageNumber", pages.getPageNumber());
    model.addAttribute("articleName", articleName);
    model.addAttribute("articleAuthors", articleAuthors);
    model.addAttribute("articleId", articleId);
    model.addAttribute("menuFlag", Constants.ARTICLE_MENU_FLAG);
    return "article/index";
}
```

2. 预览功能

预览功能是在创建文章或者修改文章后使用的，在博客后台系统中直接查询文章在博客内的具体展示效果，其实就是在 JavaScript 中加入一个跳转方法，如代码清单 14-11 所示。

代码清单 14-11　博客后台项目-文章管理预览功能前台实现

```
//预览
function  view(articleId){
   window.open("https://localhost:10000/"+articleId,"_blank");
}
```

3. 启用和禁用

启用和禁用其实就是修改当前文章的 isEnable 属性，将它的值改为 1，在前台发起 Ajax 请求，发送文章 ID 和需要修改属性的值。内容很简单，这里就不介绍了。

4. 删除文章

删除文章就是从前台将文章 ID 传入后台，然后在后台调用 Repositrory 的 deleteById()方法。

5. 新增和修改

单击文章列表页的“修改”按钮会跳转到文章编辑页，同时带出对应文章的内容。单击新增文章就会新增一篇新的文章，跳转页面时首先判断是否传了文章 ID，如果传了，就查询文章内容返回前台；如果没传，就返回一个文章 ID 为 0 的空对象。跳转方法如代码清单 14-12 所示。

代码清单 14-12　博客后台项目-文章管理新增或修改页面跳转方法

```
@GetMapping("/saveOrUpdatePage")
public String saveOrUpdateArticlePage(Model model, Long articleId) {
    Article article = new Article();
    if (articleId != null) {
        article = articleService.findArticleByArticleId(articleId);
    } else {
        article.setArticleId(0L);
    }
    model.addAttribute("article", article);
```

```
        model.addAttribute("menuFlag", Constants.ARTICLE_MENU_FLAG);
        return "article/edit";
    }
```

在文章编辑页中设置了预览文章效果，不过这里只支持 Markdown 格式预览，使用的是 showdown.js。不过有一个缺点，表格展示不出来，如果读者发现有更好的方式，那么可以自行修改。表单校验使用的是 jquery.validate.js，提供了一些非空校验等。

前台保存方法首先触发表单校验，然后调用 Ajax 发起请求。保存按钮触发方法如代码清单 14-13 所示。

代码清单 14-13　博客后台项目-文章管理保存或修改文章前台内容

```
    function saveArticle(){
        if(!$('#article_form').valid()){
          return false;
        }
        var url = getRootPath_dc() + "article/saveOrUpdate";
        var articleName = $('#articleName').val();
        var articleAuthors = $('#articleAuthors').val();
        var tagsStr = $('#tagsStr').val();
        var isTop = $('input[name="isTop"]:checked').val();
        var articleContent = $('#articleContent').val();
        var articleReadingTime = $('#articleReadingTime').val();
        var isEnable = $('#isEnable').val();
        var articleInputUser = $('#articleInputUser').val();

        if(articleId == 0){
            articleId = null;
        }
        $.ajax({
          type : "POST",
          url : url,
          dataType : "text",
          contentType: "application/json;charset=UTF-8",
          data : JSON.stringify({
            "articleId":articleId,
            "articleName": articleName,
            "articleAuthors": articleAuthors,
            "tagsStr": tagsStr,
            "isTop": isTop,
            "articleContent":articleContent,
            "articleReadingTime":articleReadingTime,
            "isEnable":isEnable,
            "articleInputUser":articleInputUser
```

```
        }),
        success: function() {
            alert("提交成功！");
        },
        error:function(){
            alert("error");
        }
      });
}
```

保存和新增后台方法会根据标签字符串内的逗号进行分割，每一个逗号前代表一个不同的标签名称，然后根据标签名称进行查询，如果不存在当前标签名称，就新增一个，并且将标签和文章进行关联，完整内容如代码清单 14-14 所示。

代码清单 14-14　博客后台项目-文章管理保存或修改文章 Service 层内容

```
@Override
@Transactional(rollbackFor = Throwable.class)
public void saveOrUpdateArticle(Article article) {
    String tagsStr = article.getTagsStr();
    List<Tag> tagList = new ArrayList<>();
    if (StringUtils.isNotBlank(tagsStr)) {
        String[] tagNames = tagsStr.split(",");
        for (String tagName : tagNames) {
            Tag tag = tagService.findTagByTagName(tagName);
            if (tag == null) {
                tag = new Tag(tagName);
                tag.setTagInputDate(new Date());
            }
            tag = tagService.saveTag(tag);
            tagList.add(tag);
        }
    }
    article.setTagList(tagList);
    if (article.getArticleId() == null) {
    article.setIsEnable(Constants.NO);
        article.setArticleInputDate(new Date());
        article.setArticleInputUser(1L);
        article.setArticleReadingTime(0);
    }
    articleRepository.save(article);
}
```

有关文章的操作大致就这些，读者可以查看配套源代码学习。

14.8 辅助功能

接下来介绍博客后台系统的辅助功能，如拦截器、定时器、认证和授权、工具类。

14.8.1 拦截器

在拦截器中首先判断是否返回页面，如果返回页面，就渲染一些基本数据，如菜单集合、头部信息、底部信息等。完整内容如代码清单 14-15 所示。

代码清单 14-15 博客后台项目-拦截器内容

```
@Component
public class RequestInterceptor extends HandlerInterceptorAdapter {
    @Autowired
    private AuthenticationService authenticationService;
    @Autowired
    private WebsiteConfigService websiteConfigService;
    @Autowired
    private TagService tagService;
    @Autowired
    private ArticleService articleService;
    @Autowired
    private MessageService messageService;
    @Autowired
    private UserService userService;

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        if (modelAndView != null) {
            ModelMap modelMap = modelAndView.getModelMap();
            Date date = new Date();
            //加载数据
            Authentication auth = authenticationService.getAuthentication();
            if (auth != null) {
                String username = auth.getName();
                modelMap.addAttribute("username", username);
                User user = userService.findByUsername(username);
                if(user != null){
                    //赋值用户所拥有的菜单集合，动态渲染菜单
```

```
                    modelMap.addAttribute("userRoleList", Constants.
getUserRoleList(user.getRoleList()));
                 }
             }
             Integer messageCount = messageService.countByIsRead
(Constants.NO);
             modelMap.addAttribute("messageCount", messageCount);
             Integer tagCount = tagService.countByTagInputDate(date);
             modelMap.addAttribute("tagCount", tagCount);
             Integer articleCount = articleService.countByArticleInputDate
(date);
             modelMap.addAttribute("articleCount", articleCount);
             modelMap.addAttribute("sumCount", articleCount + tagCount +
messageCount);
             List<Message> messageList = messageService.findAllByIsRead
(Constants.NO);
             modelMap.addAttribute("mainbarMessageList", messageList);
             modelMap.addAttribute("mainbarMessageListCount", messageList.
size());
             modelMap.addAttribute("websiteConfig", websiteConfigService.
findWebsiteConfig());

          }
       }
    }
```

14.8.2　定时器

定时器主要用于查询一些博客的数据，然后将数据封装起来，以邮件的形式发送。完整内容如代码清单 14-16 所示。

代码清单 14-16　博客后台项目-定时器内容

```
    @Component
    public class WebSiteTimer {
        private final Logger logger = LoggerFactory.getLogger(this.getClass());
        @Autowired
        private WebsiteConfigService websiteConfigService;
        @Autowired
        private JavaMailSender javaMailSender;
        @Autowired
        private WebsiteAccessService websiteAccessService;
        @Autowired
```

```
        private MessageService messageService;
        @Scheduled(cron = "0 0 0 1/1 * ?")
        private void sendDailyData() {
            String subject = "博客每日数据";
            String text = this.initData();
            WebsiteConfig websiteConfig = websiteConfigService.
findWebsiteConfig();
            SimpleMailMessage simpleMailMessage = new SimpleMailMessage();
            simpleMailMessage.setFrom(websiteConfig.getEmailUsername());
            simpleMailMessage.setTo(websiteConfig.getEmailUsername());
            simpleMailMessage.setSubject(subject);
            simpleMailMessage.setText(text);
            try {
                javaMailSender.send(simpleMailMessage);
                logger.info("发送博客每日数据成功！");
            } catch (Exception e) {
                logger.error("发送博客每日数据异常！", e);
            }
        }

        private String initData(){
            StringBuffer stringBuffer = new StringBuffer();
            SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd");
            WebsiteAccess websiteAccess = websiteAccessService.getByAccessDateIs
(new Date());
            stringBuffer.append("日期是：");
            stringBuffer.append(sdf.format(websiteAccess.getAccessDate()));
            stringBuffer.append("访问量为：");
            stringBuffer.append(websiteAccess.getAccessCount());
            stringBuffer.append("未读消息有：");
            int count = messageService.countByIsRead(Constants.YES);
            stringBuffer.append(count);
            stringBuffer.append("条");
            return stringBuffer.toString();
        }
    }
```

14.8.3 认证和授权

由于这是一个后台系统，因此不允许没有权限的人随意登录，并且在博客后台系统中默认设置了 4 个角色，分别说明如下。

- USER：拥有首页、文章管理、标签管理的权限。

- ADMIN：拥有首页、文章管理、标签管理、友情链接管理、用户管理、消息管理的权限。
- SYSADMIN：拥有首页、系统管理的权限。
- SUPERADMIN：拥有全部权限。

SecurityConfig 配置如代码清单 14-17 所示。

代码清单 14-17　博客后台项目-安全配置内容 1

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private MyUserDetailsService myUserDetailsService;

    /**
     * 权限设定
     * 分为 4 种角色：
     * USER(拥有首页、文章管理、标签管理权限)
     * ADMIN(拥有首页、文章管理、标签管理、友情链接管理、用户管理、消息管理权限)
     * SYSADMIN(拥有首页、系统管理权限)
     * SUPERADMIN(拥有全部权限)
     * 后期可以修改为动态获取
     */
    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
                .csrf().disable()//关闭 scrf
                .authorizeRequests()
                .antMatchers("/static/**").permitAll()
                .antMatchers("/").hasAnyRole("USER","ADMIN","SYSTEMADMIN",
"SUPERADMIN")
                .antMatchers("/index/**").hasAnyRole("USER","ADMIN",
"SYSTEMADMIN","SUPERADMIN")
                .antMatchers("/article/**").hasAnyRole("USER","ADMIN",
"SUPERADMIN")
                .antMatchers("/tag/**").hasAnyRole("USER","ADMIN",
"SUPERADMIN")
                .antMatchers("/link/**").hasAnyRole("ADMIN","SUPERADMIN")
                .antMatchers("/user/**").hasAnyRole("ADMIN","SUPERADMIN")
                .antMatchers("/message/**").hasAnyRole("ADMIN","SUPERADMIN")
                .antMatchers("/system/**").hasAnyRole("SYSTEMADMIN",
"SUPERADMIN")
                .and()
                .formLogin().loginPage("/login").failureUrl("/login_error").
successForwardUrl("/")
                .and()
```

```
                .logout().logoutUrl("/logout").logoutSuccessUrl("/login").
deleteCookies("JSESSIONID")
                .and()
                .exceptionHandling().accessDeniedPage("/login");
    }

    @Bean
    public static NoOpPasswordEncoder passwordEncoder() {
        return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
    }

    //根据用户名和密码实现登录
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder
authenticationManagerBuilder) throws Exception {

authenticationManagerBuilder.userDetailsService(myUserDetailsService);
    }
}
```

创建一个 MyUserDetailsService 配置用户的认证，如代码清单 14-18 所示。

代码清单 14-18　博客后台项目-安全配置内容 2

```
@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    private  UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null){
            throw new UsernameNotFoundException("用户不存在！");
        }else if(user != null && Constants.NO.equals(user.getIsEnable())){
            throw new UsernameNotFoundException("用户未启用，请联系管理员！");
        }
        List<SimpleGrantedAuthority> simpleGrantedAuthorities = new
ArrayList<>();
        for (Role role : user.getRoleList()) {
            simpleGrantedAuthorities.add(new SimpleGrantedAuthority
(role.getRoleName()));
        }
```

```
        return new org.springframework.security.core.userdetails.User
(user.getUsername(), user.getPassword(), simpleGrantedAuthorities);
    }
}
```

14.8.4　工具类

案例中的工具类只做了传入日期和天数来查询几天前的日期，如代码清单 14-19 所示。

代码清单 14-19　博客后台项目-计算日期工具类内容

```
public class DateUtils {
    public static Date getDateBefore(Date date,int day){
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(date);
        calendar.set(Calendar.DATE, calendar.get(Calendar.DATE) - day );
        return calendar.getTime();
    }
}
```

14.8.5　初始化方法

初始化方法中，插入角色信息以及默认插入一个用户名为 admin、密码为 123 的用户，如代码清单 14-20 所示。

代码清单 14-20　博客后台项目-初始化数据内容

```
@Component
public class InitData {
    @Autowired
    private RoleRepository roleRepository;
    @Autowired
    private UserService userService;
    @Autowired
    private WebsiteConfigService websiteConfigService;

    @PostConstruct
    private void initRoleData() {
        Role role1 = new Role(1L, "ROLE_USER", 1);
        Role role2 = new Role(2L, "ROLE_ADMIN", 1);
        Role role3 = new Role(3L, "ROLE_SUPERADMIN", 1);
        Role role4 = new Role(4L, "ROLE_SYSTEMADMIN", 1);
        List<Role> roleList = new ArrayList<>();
```

```
            roleList.add(role1);
            roleList.add(role2);
            roleList.add(role3);
            roleList.add(role4);
            roleRepository.saveAll(roleList);
            User user = userService.findUserByUserId(1L);
            if (user == null) {
                userService.saveOrUpdateUser(new User(1L, "admin", "123",
"admin@springboot.cn", 1, roleList));
            }
            WebsiteConfig websiteConfig = websiteConfigService.
findWebsiteConfig();
            if (websiteConfig == null) {
                websiteConfigService.saveWebsiteConfig(new WebsiteConfig(1L,
"SpringBoot 博客", "dalaoyang", 7L, "辽 ICP 备 17014944 号-1", "Dalaoyang.cn",
"dalaoyang@aliyun.com"));
            }
        }
    }
```

14.9 小　　结

本章对博客后台系统的制作进行了介绍，在实际项目中大多如此，希望大家多多练习，熟能生巧，将所学的知识运用到实践中。

参 考 文 献

[1] Spring 官网：https://spring.io/。
[2] RabbitMQ 官网：http://www.rabbitmq.com/。
[3] RocketMQ 官网：http://rocketmq.apache.org/。
[4] Kafka 官网：http://kafka.apache.org/。
[5] Elastic 官网：http://www.elastic.com/。
[6] Log4j 官网：https://logging.apache.org/log4j/2.x/。
[7] 阿里中间件团队博客：http://jm.taobao.org/。